COMP3322 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 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 Pug template engine and MongoDB. The Web service allows retrieving, adding, updating and deleting commodities from a MongoDB database. The HTML page provides an interface for displaying commodities, adding, updating and deleting them.

Set Up Runtime Environment and Install MongoDB

Follow the instructions (Steps 1 to 3) in setup_nodejs_runtime_and_examples.pdf to set up Node.js runtime environment and create an Express project named "**lab6**". Remember to replace app.set('view engine', 'jade'); in the generated **app.js** by app.set('view engine', 'pug');.

In addition, we will need a MongoDB database to store reports information. We install the database 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: Go to https://www.mongodb.org/ and download the latest version of MongoDB (choose the latest "Community Server" release to download). If you use HW311/312's 32-bit computer to complete this lab exercise, you should download the 32bit version here: https://www.mongodb.org/dl/win32/i386). Install MongoDB to a directory at your choice.

Step 3: **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)

If you use a 64-bit MongoDB on your own computer, please use the following command:

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

If you use a 32-bit MongoDB, please use the following command instead:

```
./bin/mongod  --storageEngine=mmapv1 --dbpath YourPath/lab6/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.

Step 4: **Launch the 3rd terminal**, switch to the directory where mongodb is installed, and execute the following commands:

```
./bin/mongo
use lab6
db.commodities.insert({'category':'Computer','name':'lenovo','status':'in stock'})
```

The “use lab6” command creates a database named “lab6”. The next command followed by “use lab6” inserts a new document into the “commodities” collection in the database.

After you run the insert command, 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** using a text editor. Add dependencies for MongoDB. The complete file should look like this (you do not need to worry if the version of other modules included in the **package.json** that you have generated is lower):

```
{
  "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",
    "pug": "",
    "morgan": "~1.7.0",
    "serve-favicon": "~2.3.0",
    "mongodb": "^2.2.35",
    "monk": "^3.1.3"
  }
}
```

Then install the dependencies using the terminal as follows:

```
cd lab6
npm install
```

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

Lab Exercise 1: Create the Home 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 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 for explanations of 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/3.3.1/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 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), the jQuery library on Google server, and 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 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:

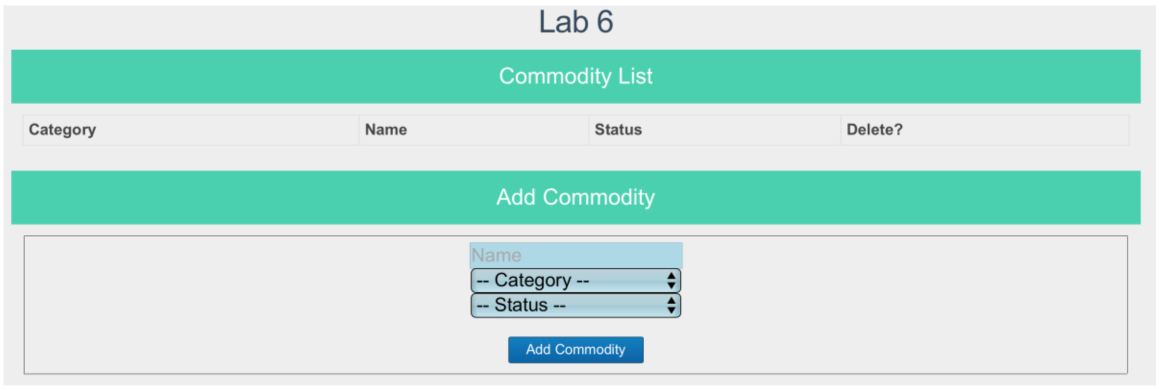


Fig. 1

Lab Exercise 2: List Commodity

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

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.

```
// 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 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 CommodityList.
 */
router.get('/commodities', function(req, res)
   { var db = req.db;
   var collection = db.get('commodities');
   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/commodities". The middleware will first retrieve the database connection. Then it will retrieve the 'commodities' 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/commodities on your browser. The browser should display a JSON response text like this:

[{"_id":"5a02a80d93677a68e4fbe8b6","category":"Computer","name":"Lenovo","status":"
in stock"}]

We can see that a "_id" attribute was added by the database server into each commodity document that we inserted earlier, which is used to uniquely identify the document in a collection. When a commodity document 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 commodity 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**:

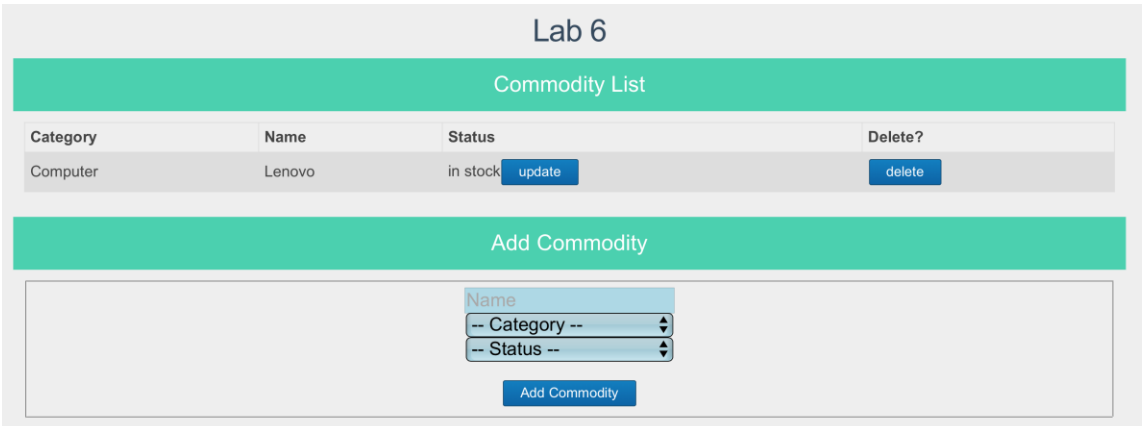
```
// DOM Ready
=============================================================
$(document).ready(function () {
   // Populate the commodity list on initial page load
   populateCommodityList();
});

// Functions =============================================================
// Fill commodity list with actual data
function populateCommodityList() {
   // Empty content string
   var listCommodity = '<table >
<tr><th>Category</th><th>Name</th><th>Status</th><th>Delete?</th></tr>';

  // jQuery AJAX call for JSON
   $.getJSON('/users/commodities', function (data) {
      // Put each item in received JSON collection into a <tr> element
      $.each(data, function () {
         listCommodity += '<tr><td>' + this.category + '</td><td>'+this.name+'</td><td
id="status_' + this._id + '">' + this.status + '<button data="' + this._id + '"
class="myButton" onclick="showStatusOptions(event,this)">update</button>'+
'</td><td>'+'<button data="' + this._id + '" class="myButton"
onclick="deleteCommodity(event,this)">delete</button>'+'</td></tr>';
      });
      listCommodity += '</table>'

        // Inject the whole commodity list string into our existing #commodityList element
      $('#commodityList').html(listCommodity);
   });
};
```

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 commodity list. You should see that the commodity that we inserted into the database earlier is now displayed on the web page:



Lab 6

Commodity List

Category	Name	Status	Delete?
Computer	Lenovo	in stock update	delete

Add Commodity

Fig. 2

Lab Exercise 3: Add a New Commodity

We next implement the server-side and client-side code for adding a new commodity document 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/addcommodity .

```
/*
* POST to add commodity
*/
router.post('/addcommodity', function (req, res) {
 var db = req.db;
 var collection = db.get('commodities');

 //insert new commodity document
 collection.insert(req.body, function (err, result) {
       res.send(
         (err === null) ? { msg: '' } : { msg: err }
       );
 });
});
```

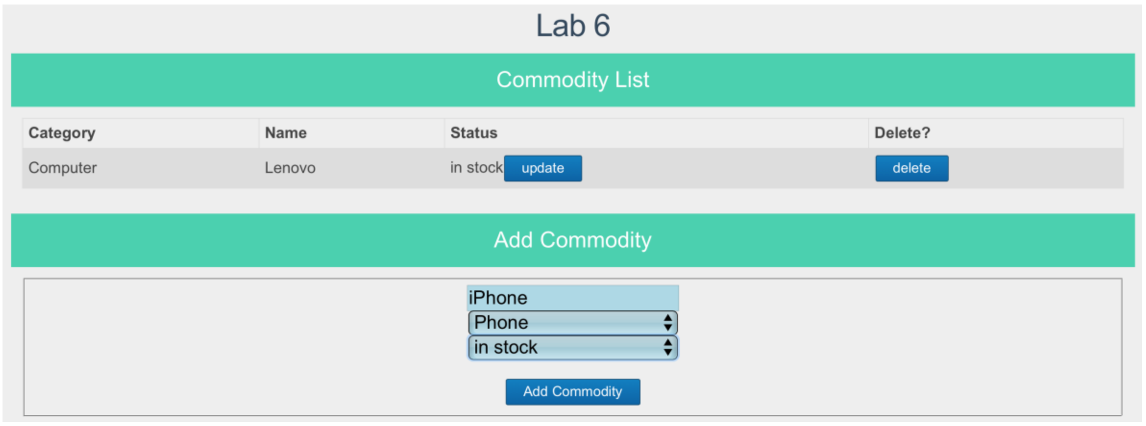
You do not need to worry about inserting duplicate documents with the same category, name, and/or status. Their _id values will be different in the MongoDB database.

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 Commodity" button is clicked, the **addCommodity** function will be invoked. **addCommodity** first checks if all fields in the "**#addcommodity**" division have been filled: if not, it prompts 'Please fill in all fields' and return; otherwise, it sends an AJAX HTTP POST request to http://localhost:3000/users/addcommodity, carrying a JSON string containing the input information of the new commodity inside its body. Upon receiving a success HTTP response, the client clears all the fields in the "**#addcommodity**" division, and updates the commodity list by calling **populateCommodityList ()**.

```
// Add Commodity button click
$('#btnAddcommodity').on('click', addCommodity);
```

```
// Add commodity
function addCommodity(event) {
  event.preventDefault();
  //validation - increase errorCount if any field is blank
  var errorCount = 0;
  $('#addcommodity input').each(function (index, val) {
    if ($(this).val() === '') { errorCount++; }
  });
  $('#addCommodity select').each(function (index, val) {
    if ($(this).val() === '') { errorCount++; }
  });
  // Check and make sure errorCount's still at zero
  if (errorCount === 0) {
    // If it is, compile all commodity information into one object
    var category = $('#addcommodity fieldset select#inputCategory').val();
    var name = $('#addcommodity fieldset input#inputName').val();
    var status = $('#addcommodity fieldset select#inputStatus').val();
    var newCommodity = {
      'category': category,
      'name': name,
      'status': status
    }
    $.ajax({
      type: 'POST',
      data: newCommodity,
      url: '/users/addcommodity',
      dataType: 'JSON'
    }).done(function (response) {
      // Check for successful (blank) response
      if (response.msg === '') {
        // Clear the form inputs
        $('#addcommodity fieldset input').val('');
        $('#addcommodity fieldset select').val('0');
        // Update the table
        populateCommodityList();
      }
      else {
        // If something goes wrong, alert the error message that our service returned
        alert('Error: ' + response.msg);
      }
    });
  }
  else {
    // If errorCount is more than 0, prompt to fill in all fields
    alert('Please fill in all fields');
    return false;
  }
};
```

Step 3: Restart your Express app and browse http://localhost:3000 again. Add information of a new commodity as Fig. 3 below. After clicking the “Add Commodity” button, you should see a page as shown in Fig. 4.


Lab 6

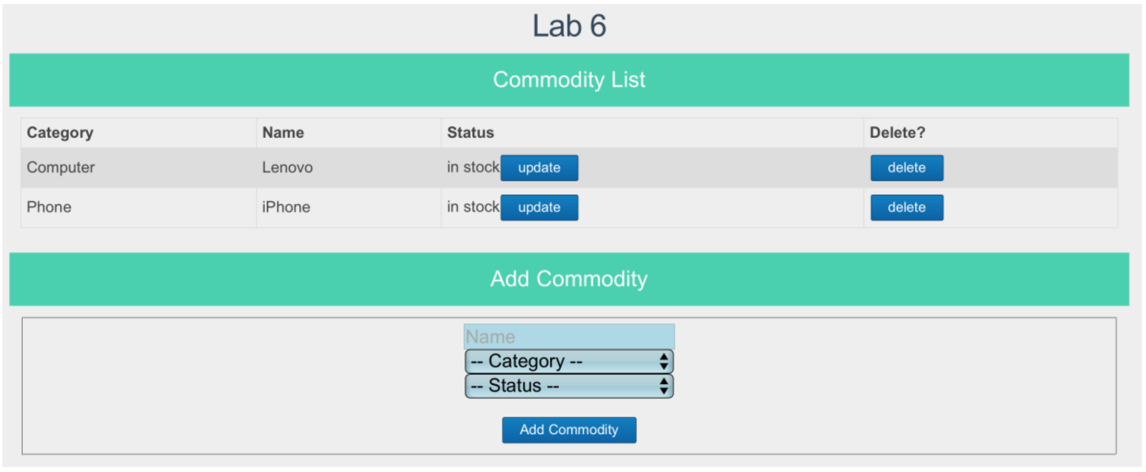
Commodity List

Category	Name	Status	Delete?
Computer	Lenovo	in stock update	delete

Add Commodity

iPhone
Phone
in stock
Add Commodity

Fig. 3


Lab 6

Commodity List

Category	Name	Status	Delete?
Computer	Lenovo	in stock update	delete
Phone	iPhone	in stock update	delete

Add Commodity

Name
-- Category --
-- Status --
Add Commodity

Fig. 4

Lab Exercise 4: Delete a Commodity

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

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

```
/*
 * DELETE to delete a commodity.
 */
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 commodity document is saved to the **"data"** attribute of a delete **"<button>"** element. The client will send an AJAX HTTP DELETE request to the following URL once you click the "**delete**" button:

http://localhost:3000/users/deletecommodity/xx
(replace xx by the value of "_id" attribute of a commodity document to be deleted).

2. The middleware should handle HTTP DELETE requests for path

/deletecommodity/: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 commodity document from the commodities 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.

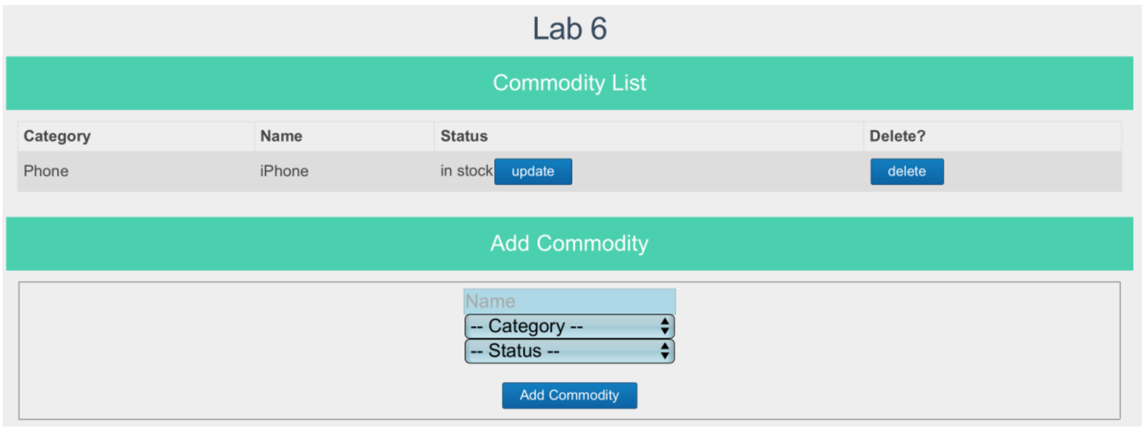
```
// Delete Commodity
function deleteCommodity(event, instance) {
   event.preventDefault();
   var id = $(instance).attr('data');
   $.ajax({
       ?
   }).done(function (response) {
       ?
   });
};
```

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

2. Upon successful deletion, you should refresh the "Commodity List" display on the web page; otherwise, prompt 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:


Lab 6

Commodity List

Category	Name	Status	Delete?
Phone	iPhone	in stock update	delete

Add Commodity

Fig. 7 After clicking "delete" button in the row of "Computer Lenovo in stock"

Lab Exercise 5: Update a Commodity

In this part, we implement the server-side and client-side code for updating the status of an existing commodity document in the database.

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

```
/*
* PUT to update a commodity (status)
*/
router.put('/updatecommodity/:id', function (req, res) {
  var db = req.db;
  var collection = db.get('commodities');
  var commodityToUpdate = req.params.id;
  var newStatus = req.body.status;

//TO DO: update status of the commodity in commodities collection, according to
commodityToUpdate and newStatus

});
```

Implement the code in the above middleware, for updating the "status" of an existing commodity document in the commodities collection, whose "_id" is carried in the URL of the PUT request message, to the new status carried in request body. **Hint**: use **collection.update()** (https://automattic.github.io/monk/docs/collection/update.html). Upon successful update, 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.

```
// Show Status Selection
function showStatusOptions(event,instance) {
   event.preventDefault();
   var id = $(instance).attr('data');

   var statusField='<select><option value="0">-- Status --</option><option value="in
stock">in stock</option><option value="out of stock">out of
stock</option></select><button data="' + id + '" class="myButton"
onclick="updateCommodity(event,this)">update</button>';

   $("#status_"+id).html(statusField);
};

// Update Commodity (status)
function updateCommodity(event,instance) {
   event.preventDefault();
   var id = $(instance).attr('data');

   var newStatus = $("#status_"+id + " select").val();

   if (newStatus === '0'){
      alert('Please select status');
      return false;
   }
  else{
         var changeStatus = {
            'status': newStatus
```

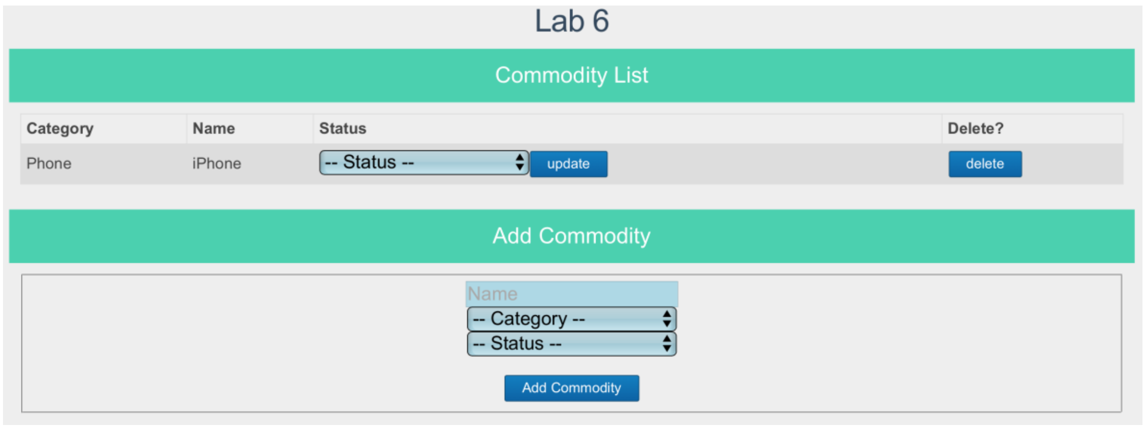
```
                }

                $.ajax({
                   type: ?,
                   url: ?
                   data: ?
                   dataType: 'JSON'
                }).done(function (response) {
                   if (response.msg === '') {
                      ?
                   }
                   else {
                      ?
                   }
                });
            }
     };
```

Note that in the following code we have added in Step 4 of Lab Exercise 2, in the <td> element where we display status of a commodity document, we also include a **“<button>”** element, i.e., the “**update**” button as shown in previous screenshots.

```
listCommodity += '<tr><td>' + this.category + '</td><td>'+this.name+'</td><td
id="status_' + this._id + '">' + this.status + '<button data="' + this._id + '"
class="myButton" onclick="showStatusOptions(event,this)">update</button>'+
'</td><td>'+'<button data="' + this._id + '" class="myButton"
onclick="deleteCommodity(event,this)">delete</button>'+'</td></tr>';
```

When this button is clicked, showStatusOptions() is invoked, which displays a <select> element for status selection and an “update” button in the cell (see Fig. 8). Note the HTML code of this update button is different from the previous update button.



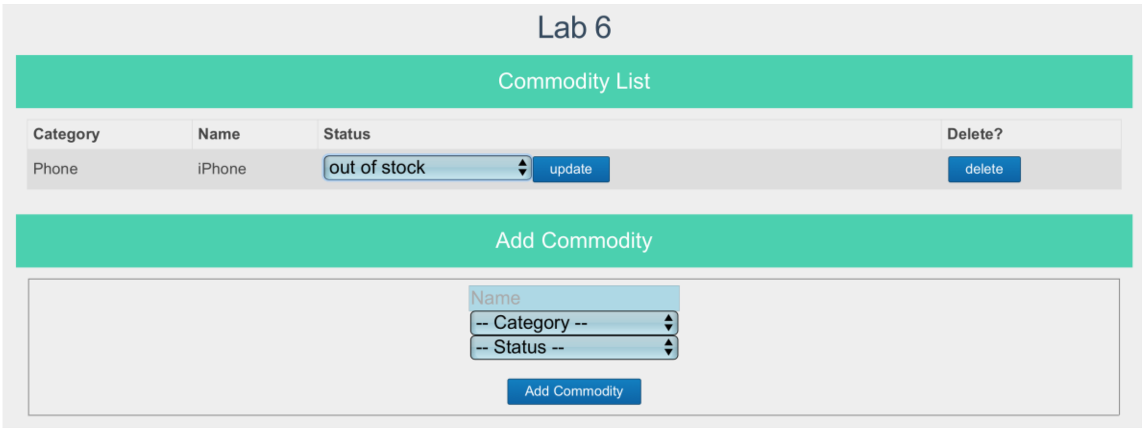
Category	Name	Status	Delete?
Phone	iPhone	-- Status -- update	delete

Fig. 8 After clicking “update” button in the row of “Phone iPhone in stock”

When the update button in the above page view is clicked, updateCommodity() function is invoked. It checks if a status has been selected: if no, it prompts “Please select status”; otherwise, it sends an HTTP PUT request to http://localhost:3000/users/updatecommodity/xx (replace xx by the value of “_id” of the commodity to be updated).

Replace "?" with correct code to finish the client-side code for sending an AJAX HTTP PUT request and handling the response. Especially, upon successful update, you should refresh the "Commodity List" display on the web page; otherwise, prompt the error message carried in the response using alert(). (Note: You do not need to handle the case that the newly selected status is in fact the same as the old status.)

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



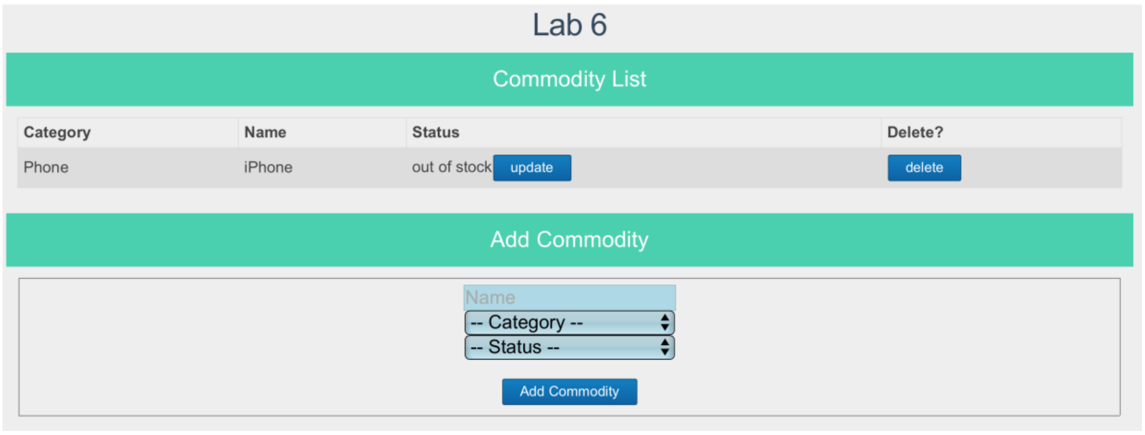
Lab 6

Commodity List

Category	Name	Status	Delete?
Phone	iPhone	out of stock update	delete

Add Commodity

Fig. 9 After selecting "out of stock"



Lab 6

Commodity List

Category	Name	Status	Delete?
Phone	iPhone	out of stock update	delete

Add Commodity

Fig. 10 After clicking "update" on the Fig. 9 view

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 lab6.zip to i.cs.hku.hk web server under "**lab6**" before **23:59 Sunday April 1, 2018**. The URL to access your submission should be **http://i.cs.hku.hk/~[your_CSID]/lab6/lab6.zip**. We will check your files for lab 6 marking.