COMP3322 Modern Technologies on World Wide Web

Workshop – RESTful Web Service Using Express.js, MongoDB and Pug

Introduction

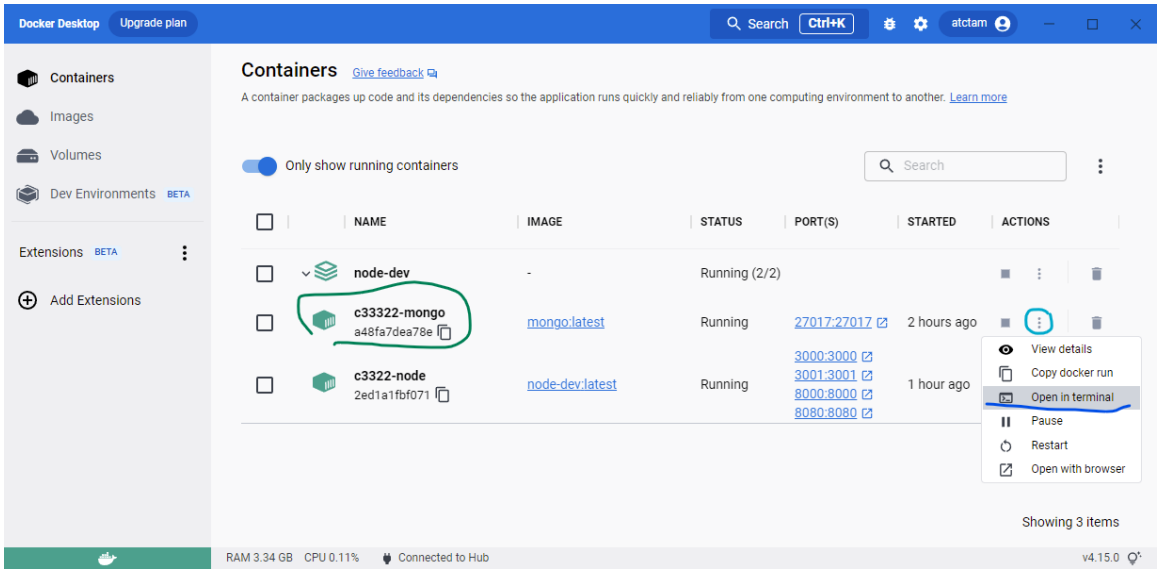
In this workshop, we will use **Node.js** to implement a RESTful Web service and the HTML content to access the Web service. In particular, we will use the Express.js web framework based on Node.js, together with the Pug template engine and **MongoDB**. The Web service allows retrieving, adding, updating and deleting commodities from a MongoDB database. The HTML page provides an interface for displaying commodities, adding, updating and deleting them.

Create a workshop5 Express project and Add data to MongoDB

Step 1: Follow the instructions in **Express-generator.pdf** to create an Express project named "**workshop5**". Remember to invoke express-generator with --view=pug to generate an app.js that uses Pug template engine.

We also need a MongoDB database to store reports information. We are going to use the c3322-mongo container for this workshop. By default, it stores all the data in the Node-dev/data/db folder.

Step 2: Open another terminal for the c3322-mongo container as follows.



Step 3: Execute the following commands:

```
mongosh
use workshop5
db.commodities.insertOne({'category':'Computer','name':'lenovo','status':'in stock'})
```

The first command – mongosh is for running a Mongo Shell, which is the command-line program to access the MongoDB database.

The "use workshop5" command creates a database named "workshop5" in our MongoDB server. The next command followed by "use workshop5" inserts a new document into the "commodities" collection in the database.

After you run the insertOne command, you should see the following output on the terminal.

```
{
    acknowledged: true,
    insertedId: ObjectId("63b82df04bb24a1ded5aa54b")
}
```

You can use the same method to insert more records into the database collection to test your program.

Step 4: Add the mongoose package to the project. Switch to the c3322-node terminal (opened in Step1) and make sure you are in workshop5 folder before entering the following command:

```
npm install mongoose
```

Exercise 1: Create the Home Page Using Pug

We next modify the pug templates and css file in "workshop5", in order to render the homepage of our Express app. You can access all files inside the "workshop5" folder by accessing the host folder "Node-dev/applications/workshop5".

Step 1:

1. Replace ./views/index.pug with index.pug which we provide in **WK5-Node.zip**. Please refer to https://pugjs.org/api/reference.html for explanations of the code in the file.
2. Replace ./public/stylesheets/style.css with style.css file we provide in **WK5-Node.zip**.

Step 2: Open ./views/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='/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 ./public/stylesheets/style.css for styling and ./public/javascripts/externalJS.js containing client-side JavaScript (which we will create under that directory in 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 ./routes/index.js and replace "express" in the line "res.render('index', { title: 'Express' });" by "Workshop5".

folder test terminal

Step 4: Now let's check out the web page rendered using the new Pug files. In the c3322-node 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.

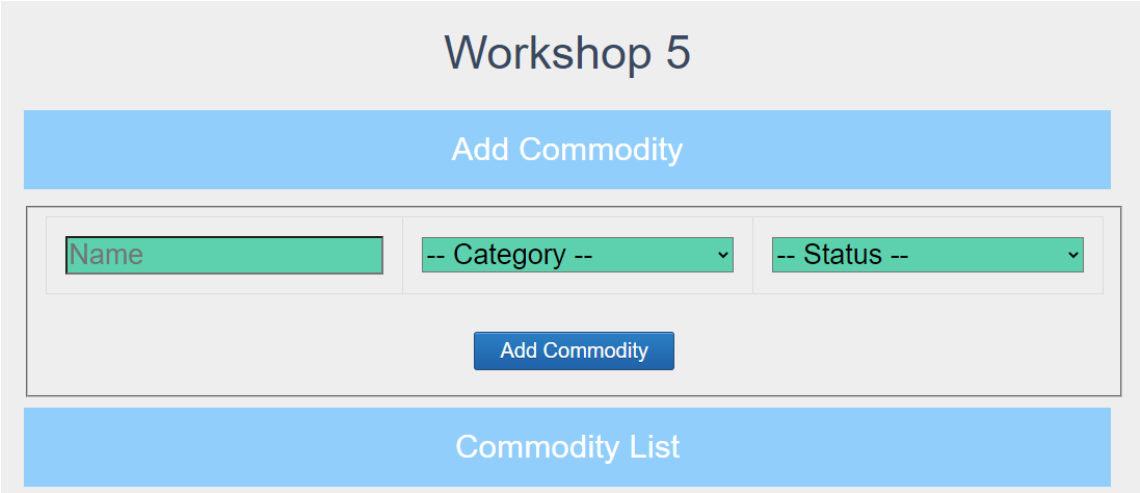


Fig. 1

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 ***before*** "var indexRouter = require('./routes/index');". By doing so, we establish a connection with the database "workshop5" that we created.

```
//Database
var db = require('mongoose');
db.set('strictQuery', false); //to avoid the warning message
db.connect('mongodb://mongodb/workshop5', {useNewUrlParser: true,
useUnifiedTopology: true})
  .then(() => {
    console.log("Connected to MongoDB");
  })
  .catch(err => {
    console.log("MongoDB connection error: "+err);
});

//Set the Schema
var mySchema = new db.Schema({
  category: String,
  name: String,
  status: String
});

//Create my model
var commodity = db.model("commodity", mySchema, "commodities");
```

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

```
// Make our model accessible to routers
app.use(function(req,res,next) {
```

```
  req.commodity = commodity;
  next();
});
```

By assigning the **commodity** object to **req.commodity**, subsequent router modules can use the **req.commodity** object to access the database.

Step 2: Now open ./routes/user.js 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) {
  //Get the data
  req.commodity.find()
    .then(docs => {
      console.log(docs);
      res.json(docs);
    })
    .catch(err => {
      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 retrieve the commodities collection via the req.commodity object from the connected database. Then it will 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 Exercise 1, we link the rendered HTML page to externalJS.js. Create an externalJS.js file under the directory ./public/javascripts. Put the following JavaScript code into externalJS.js:

```
// DOM Ready =============================================================
window.addEventListener('DOMContentLoaded', () => {
  // 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>Name</th><th>Category</th><th>Status</th><th>Delete?</th></tr>';

  //fetch call for JSON
  fetch('/users/commodities')
  .then(response => {
    response.json().then(data => {
      // Put each item in received JSON collection into a <tr> element
      data.forEach(elm => {
        listCommodity += '<tr><td>' + elm.name + '</td><td>'+
          elm.category +'</td><td id="status_' + elm._id + '">' +
          elm.status + '<button data="' + elm._id +
          '" class="myButton"
onclick="showStatusOptions(event)">update</button>'+
          '</td><td>'+'<button data="' + elm._id +
          '" class="myButton"
onclick="deleteCommodity(event)">delete</button>'+
          '</td></tr>';
      });
      listCommodity += '</table>'

      // Inject the whole commodity list string into our existing
#commodityList element
      document.querySelector('#commodityList').innerHTML=listCommodity;
    });
  });
};
```

Step 5: Now restart the express app and then 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 JS code in **externalJS.js** is executed when the page has been loaded by the browser, 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:

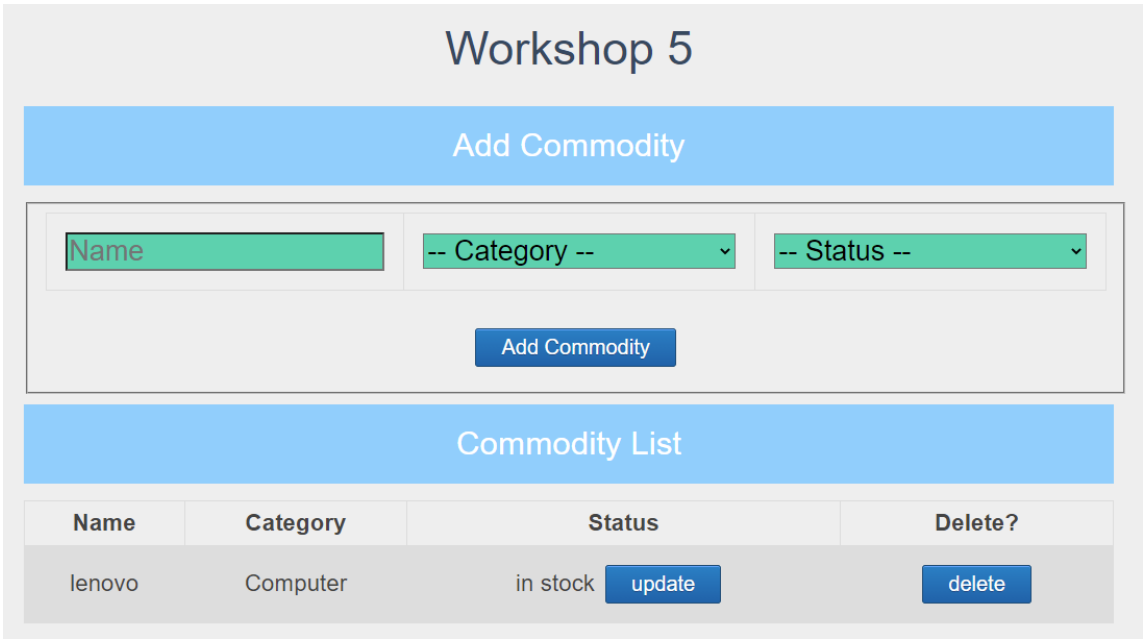


Fig. 2

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 ./routes/user.js and add the following middleware into this file (before "module.exports = router;"), which handles HTTP POST requests sent for http://localhost:3000/users/addcommodity.

```
/*
* POST to add commodity
*/
router.post('/addcommodity', function (req, res) {
  var addRecord = new req.commodity({
    category: req.body.category,
    name: req.body.name,
    status: req.body.status
  });

  //add new commodity document
  addRecord.save()
  .then(() => {res.send({ msg: '' });} )
  .catch(err => res.send({ 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 ./public/javascripts/externalJS.js 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 a fetch() 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
document.querySelector('#btnAddcommodity').addEventListener('click',
addCommodity);

// Add commodity
function addCommodity(event) {
  event.preventDefault();
  //validation - increase errorCount if any field is blank
  var errorCount = 0;
  document.querySelectorAll('#addcommodity input').forEach((elm) => {
    if (elm.value === '') { errorCount++; }
  });
  document.querySelectorAll('#addCommodity select').forEach((elm) => {
    if (elm.value === '') { 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 = document.querySelector('select#inputCategory').value;
```

```
    var name = document.querySelector('input#inputName').value;
    var status = document.querySelector('select#inputStatus').value;
    var newCommodity = {
      'category': category,
      'name': name,
      'status': status
    }
    fetch('/users/addcommodity', {method: 'POST', body:
JSON.stringify(newCommodity), headers: {'Content-Type':
'application/json'}})
    .then(response => {
      if (response.ok) {
        response.json().then(data => {
          if (data.msg === '') {
            // Clear the form inputs
            document.querySelector('input#inputName').value = '';
            document.querySelector('select#inputCategory').value = 0;
            document.querySelector('select#inputStatus').value = 0;
            // Update the table
            populateCommodityList();
          } else {
            // If something goes wrong, alert the error message that our
service returned
            alert('Error: ' + data.msg);
          }
        });
      } else {
        alert("HTTP return status: "+response.status);
      }
    });
  } 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.

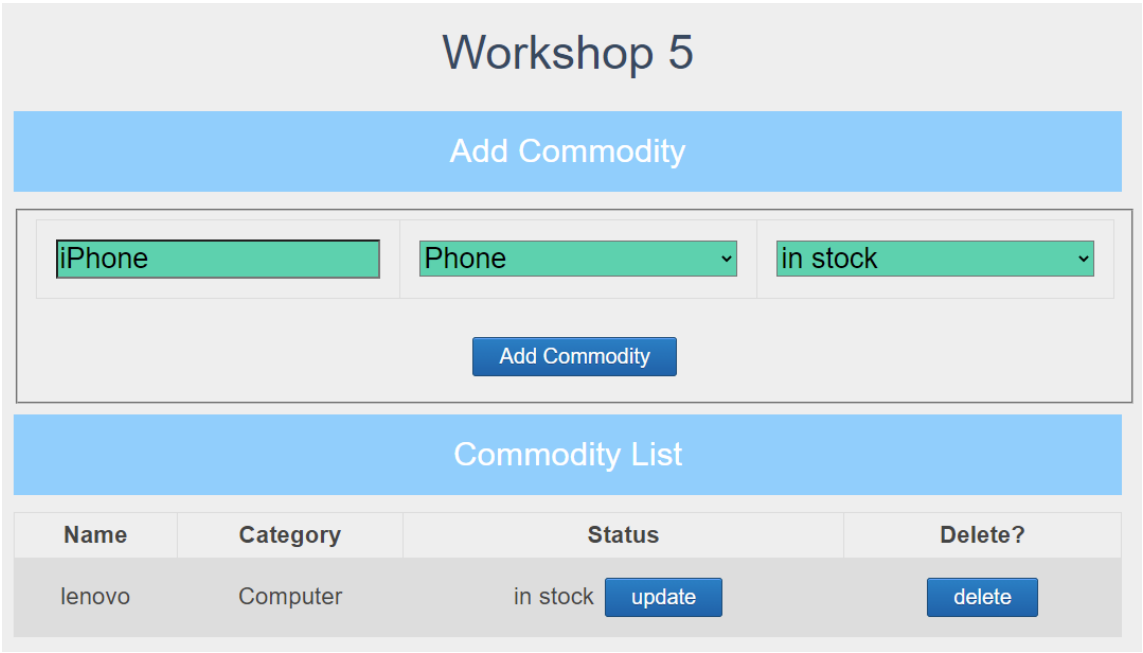


Fig. 3 Before add 'iPhone'

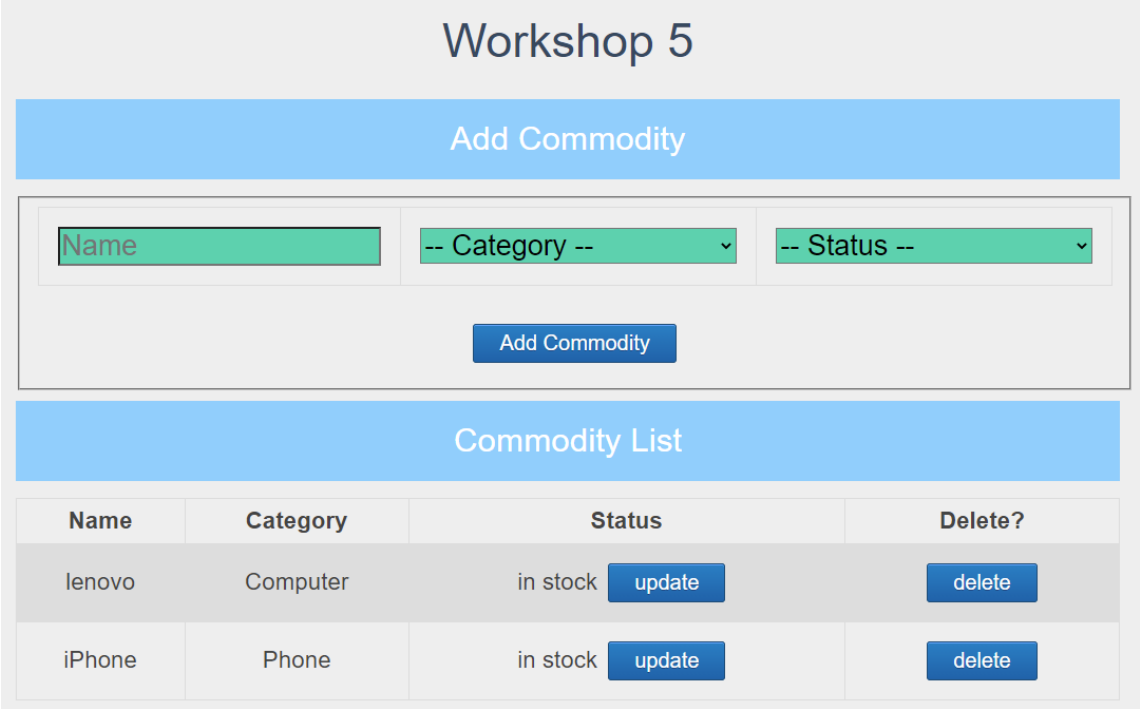


Fig. 4 After 'add iPhone'

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 ./routes/user.js 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 Exercise 2, the "_id" attribute of a commodity document is saved to the "data" attribute of a delete **<button>** element. The client will send a fetch() 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 findByIdAndDelete() method of the Mongoose API 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 ./public/javascripts/extenernalJS.js and add the following code at the end of the file.

```
// Delete Commodity
function deleteCommodity(event) {
  event.preventDefault();
  var id = event.target.getAttribute('data');
  fetch(  ?   )
  .then(response => {
    if (response.ok) {
        ?
    } else {
      alert("HTTP return status: "+response.status);
    }
  });
};
```

Replace "?" with correct code to finish the client-side code for sending a fetch() DELETE request and handling the response. You should follow these hints:

You should fill in correct **method** and **url** of the HTTP DELETE request in the fetch() method call. Upon successful deletion, you should refresh the "Commodity List" that 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:

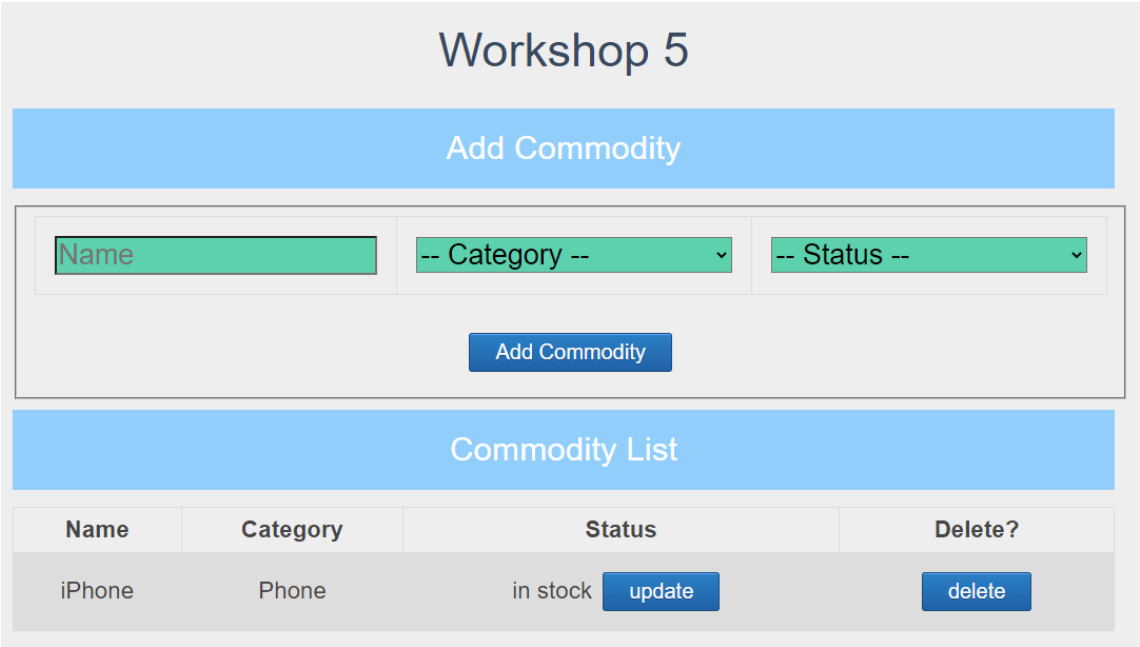


Fig. 5 After clicking "delete" button in the row of "Computer Lenovo in stock"

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 ./routes/user.js and add the following middleware:

```
/*
* PUT to update a commodity (status)
*/
router.put('/updatecommodity/:id', function (req, res) {
```

```
  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 the findByIDAndUpdate() method of the Mongoose API (https://mongoosejs.com/docs/api.html#model_Model.findByIdAndUpdate).

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 ./public/javascripts/externalJS.js and add the following code at the end of the file.

```
// Show Status Selection
function showStatusOptions(event) {
    event.preventDefault();
    var id = event.target.getAttribute('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)">update</button>';

    document.querySelector("#status_"+id).innerHTML=statusField;
};

// Update Commodity (status)
function updateCommodity(event) {
  event.preventDefault();
  var id = event.target.getAttribute('data');

  var newStatus = document.querySelector("#status_"+id + " select").value;

  if (newStatus === '0'){
       alert('Please select status');
       return false;
  } else {
    var changeStatus = {
      'status': newStatus
    }
    fetch(   ?   )
    .then(response => {
      if (response.ok) {
        response.json().then(data =>{
            ?
        });
      } else {
        alert("HTTP return status: "+response.status);
      }
    });
  }
};
```

Note that in the following code we have added in Step 4 of 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.name +
'</td><td>'+this.category+'</td><td id="status_' + this._id + '">' +
this.status + '<button data="' + this._id + '" class="myButton"
onclick="showStatusOptions(event)">update</button>'+ '</td><td>'+'<button
data="' + this._id + '" class="myButton"
onclick="deleteCommodity(event)">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.

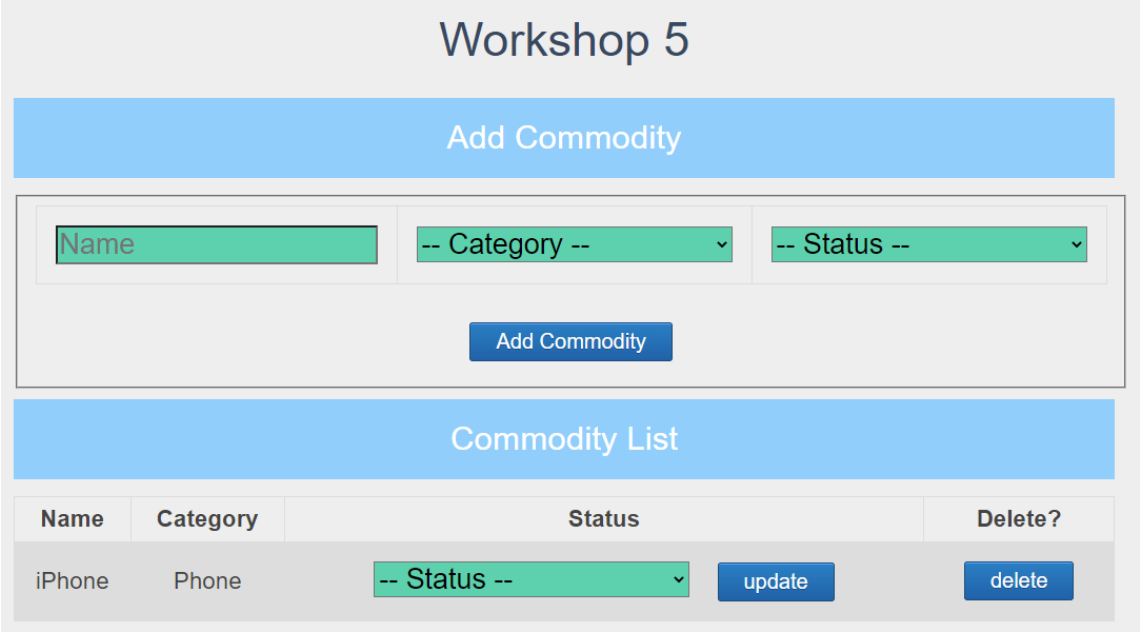


Fig. 6 After clicking "update" button in the row of "Phone iPhone in stock"

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 a fetch() request and handling the response. Especially, upon successful update, you should refresh the "Commodity List" that 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:

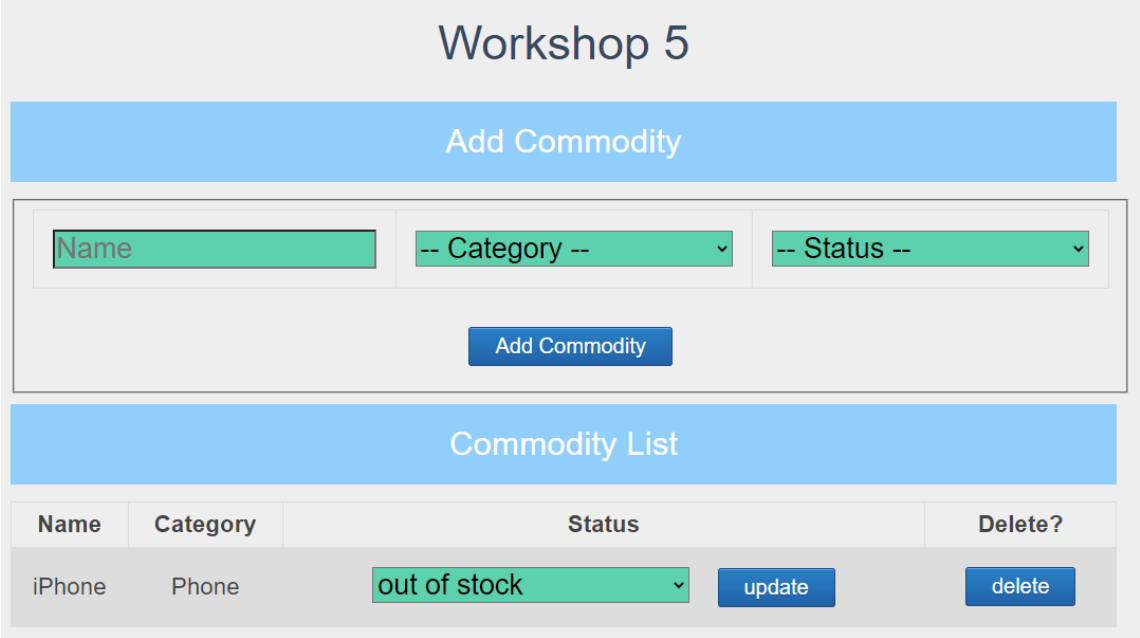


Fig. 7 After select “out of stock”

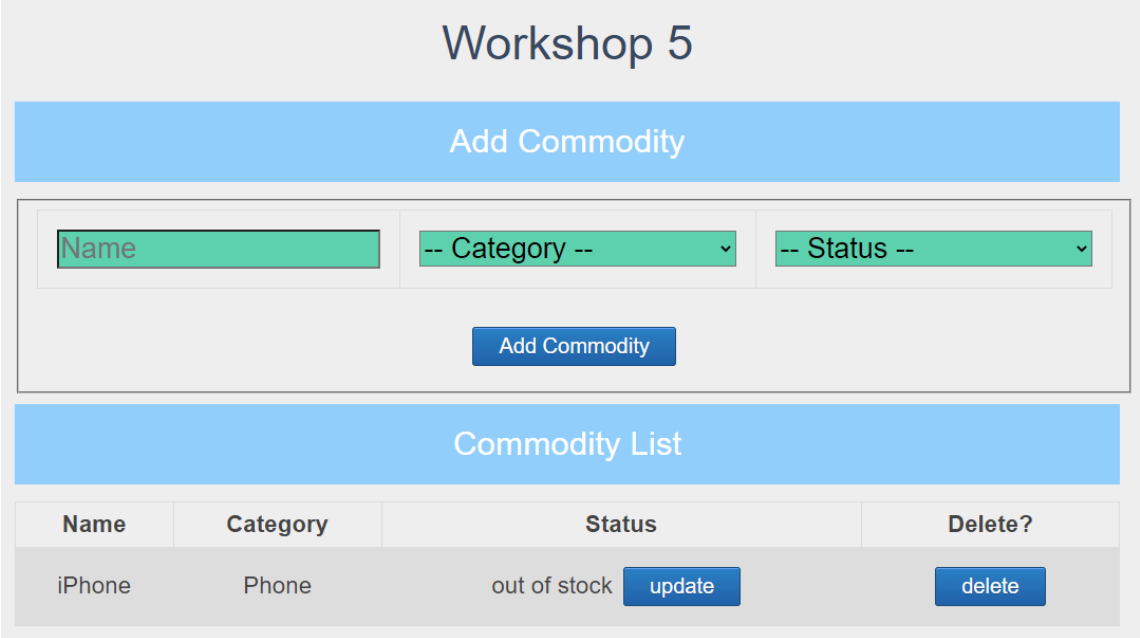


Fig. 8 After click “update” button in the Fig. 9

When you complete the project, your workshop5 directory should be in the following file hierarchy structure:

```
workshop5
├── app.js
├── bin
│   └── www
├── node_modules [many entries]
├── package-lock.json
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   │   └── externalJS.js
│   └── stylesheets
```

```
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug
```