LAB 13b

Intermediate Node

What You Will Learn

- How to make server requests using curl.
- How to create an API that implements CRUD functionality.
- How to implement file uploads.
- How to use socket.io to create a push-based chat application.

Note

This chapter's content has been split into two labs: Lab13a and Lab13b.

Approximate Time

The exercises in this lab should take approximately 60 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: January 15, 2024
Revisions: Added curl exercise, moved CRUD API exercises from Lab13a and revised them, added uploading files, removed ejs exercise, moved chat exercises from Lab13a

Preparing Directories

1 The starting `lab13b` folder has been provided for you (within the zip folder downloaded from Gumroad).

Exercise 13b.1 — Setting Up Node Project

1 Navigate to your working folder and enter the following command.

```
npm init
```

2 Install the express package via the following command.

```
npm install express
```

3 Run your API (it's the same one from Lab13a) via the following command:

```
node --env-file=config.env stocks-api
```

4 Test in browser using one of requests output in the console.

If you are using Visual Code's terminal then you can simply Ctrl-click the URLs.

Implementing CRUD Behaviors

For JavaScript intensive applications, it is common for web APIs to provide not only the ability to retrieve data (as in Lab13a), but also create, update, and delete data as well. Since REST APIs use HTTP, it is common to use different HTTP verbs to signal whether we want to create, retrieve, update, or delete (CRUD) data. While one could associate any HTTP verb with any CRUD action, it is convention to use `GET` for retrieve requests, `POST` for create requests, `PUT` for update requests, and `DELETE` for delete requests.

It can be tricky to test CRUD-based APIs, since browser requests via the address bar can only be GET requests. While an HTML form can make POST requests, there is no way to make PUT or DELETE requests without writing JavaScript. For this reason, developers often make use of specialized tools such as Postman or Insomnia to help test APIs. A free option that can be used instead is **curl**, a command line tool that lets you make requests to a server. It provides a quick way to make different types of HTTP requests without requiring a browser, an HTML form, and/or JavaScript coding.

Exercise 13b.2 — Understanding HTTP Data With Curl

1 Ensure your stocks api is currently running by running the following command, ideally from within the Visual Code Terminal.

```
node --env-file=config.env stocks-api
```

2 Open another terminal (not in Visual Code), for instance, using Git Bash.

3 Enter the following command in this second terminal:

```
curl -i http://localhost:8080/stock/amzn
```

cURL, which stands for client URL, lets you make requests to a server using a command-line interface (CLI). It provides a quick way to make different types of HTTP requests without requiring a browser, an HTML form, and/or JavaScript coding.

This command shows both the response headers and the received JSON data (which you've seen already when you've requested this URL in a browser).

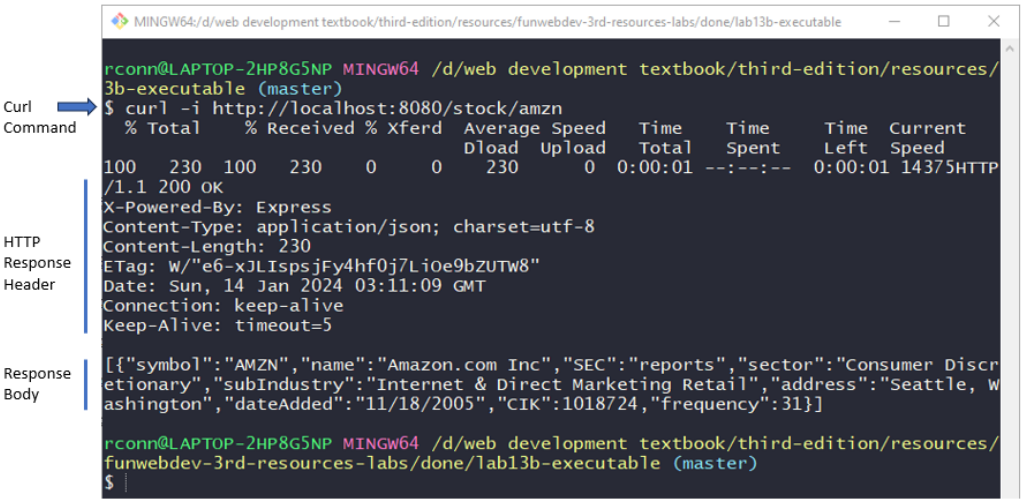

Figure 13.1 – Understanding Curl

4 In `stock-router.js`, there is already an empty router function named `handleCurlTest`, which you will use to test out different HTTP verbs. Add the following to it.

```
const handleCurlTest = (app) => {
   app.get('/test', (req,resp) => {
            resp.send('<html><em>Testing</em></html>');
   });
};
```

5 Run the code:

```
node --env-file=config.env stocks-api
```

6 In your second terminal, enter the following curl command:

```
curl -i http://localhost:8080/test
```

Again, you will see the full response: the HTTP headers and the HTML in the Step 5.

7 In `stocks-api.js`, add the following code near the top of the file:

```
const app = express();
app.use(express.urlencoded({ extended: true }));
```

This will allow your Node application to access querystring data, for instance, when a `<form>` with `method=POST` is submitted.

8 In `stock-router.js`, add the following code.

```
const handleCurlTest = (app) => {
   app.post('/test', (req,resp) => {
      resp.send(req.body);
   });

   app.get('/test', (req,resp) => {
      resp.send('<html><em>Testing</em></html>')
   });
};
```

This will display the querystring data that has been received in the request.

9 Run the node application again, and in a separate terminal enter the following (it's all on a single line, though it is split here in the word processor across multiple lines):

```
curl -i -d "p1=val1&p2=val2"
     -H "Content-Type: application/x-www-form-urlencoded"
     -X POST http://localhost:8080/test
```

This includes the querystring `p1=val1&p2=val2` in a POST request. Notice also that the request specifies the content type as `x-www-form-urlencoded`, which indicates the form data is being transmitted to the server as a query string.

10 In `stocks-api.js`, add the following code:

```
const app = express();
app.use(express.json());
```

This will allow your Node application to access JSON data sent from a browser (yes, that's JSON data sent from the browser to the server).

11 In `stock-router.js`, modify the following code.

```
const handleCurlTest = (app) => {
   app.post('/test', (req,resp) => {
      resp.send(req.body);
   });

   app.put('/test', (req,resp) => {
      resp.json(req.body);
   });
```

Notice that this handler will run when a PUT request is received.

12 Run the Node application again, and in the separate terminal enter the following:

```
curl -i -d '{"key1":"val1", "key2":"val2"}'
     -H "Content-Type: application/json"
     -X PUT http://localhost:8080/test
```

Notice that this time we are making a PUT request and sending it some sample JSON data.

13 Add the following:

```
const handleCurlTest = (app) => {
   ...
   app.delete('/test', (req,resp) => {
      resp.send('<html><em>Delete Tested</em></html>')
   });
```

14 Run the Node application again, and in the separate terminal enter the following:

```
curl -i -X DELETE http://localhost:8080/test
```

You've now added some simple handling for all four HTTP verbs (get,post,put, delete) and tested this handling by making requests using curl.

In the next set of exercises, you will add working CRUD functionality to your stocks API. A form has also been provided that makes the POST/PUT/DELETE requests. In this example, your code will simply modify the in-memory JSON array. In lab14b, you will learn how to persist received data using a database.

Exercise 13b.3 — Using HTTP Put For Updating

1 Examine `tester.html` and `tester.js` in the `public` folder.

Rather than continuing to use curl, you will use this form to test the CRUD functionality of your API. If you examine the JavaScript, you will see it uses `fetch()` to make POST, PUT, and DELETE requests of the /stock API.

2 In `stock-router.js` add the following code to the `handleSingleSymbol()` function (some code omitted).

```
const handleSingleSymbol = (stocks, app) => {
   app.get('/stock/:symbol', (req,resp) => {
     ...
   });

   // PUT request: Update specified stock using provided data
   app.put('/stock/:symbol', (req,resp) => {
      const symbolToUpd = req.params.symbol.toUpperCase();
      // find index for stock with this symbol
      let indx = stocks.findIndex(s => s.symbol === symbolToUpd);
      // if didn't find it, then return message
     if (indx < 0) {
         resp.json(jsonMessage(`${symbolToUpd} not found`));
      } else {
         // symbol found, so replace its value with form values
         stocks[indx] = req.body;
         // let requestor know it worked
         resp.json(jsonMessage(`${symbolToUpd} updated`));
      }
   });
};
```

3 Ensure your stocks api is currently running by running the following command.

```
node --env-file=config.env stocks-api
```

4 Test by requesting `static/tester.html` in the browser. It already has a prefilled in form. Edit some of the fields (but **not** the symbol field) and click the **Update** button.

5 After updating, make a GET request for the same symbol using the `View current data` link, or make a request for :

`http://localhost:8080/stock/ADBE`

It should display the updated values.

6 Our code only changed the data in the in-memory stock collection. To verify, stop (ctrl-c) the application, re-run it, and re-request the AMZN stock. It will be back to the original values.

In Lab14b, you will make changes persistent by recording them in a database.

Exercise 13b.4 — Adding Create and Delete to API

1 In `stock-router.js` add the following code to the `handleSingleSymbol()` function (some code omitted).

```
const handleSingleSymbol = (stocks, app) => {
   app.get('/stock/:symbol', (req,resp) => {
     ...
   });
   app.put('/stock/:symbol', (req,resp) => {
     ...
   });
   app.post('/stock/:symbol', (req,resp) => {
      // create a new stock, populate it with received data,
      // and then add it to the array
       stocks.push({
          symbol: req.body.symbol,
          company: req.body.company,
          sector: req.body.sector,
          sub: req.body.sub,
          address: req.body.address,
          exchange: req.body.exchange,
        });
        resp.json(jsonMessage(`New stock ${req.body.symbol}
                                  added`));
   });
   app.delete('/stock/:symbol', (req,resp) => {
      // find symbol element in array
       const symbolToDel = req.params.symbol.toUpperCase();
       let indx = stocks.findIndex((s) => s.symbol === symbolToDel);
      // if found remove it from array
       if (indx > -1) {
          stocks.splice(indx, 1);
          resp.json(jsonMessage(`${symbolToDel} deleted`));
       } else {
          resp.json(jsonMessage(`${symbolToDel} not found`));
       }
   });
```

2 Test using `tester.html`. Try deleting the current symbol, verify it has been removed from array via the test link, then add it back using the insert button.

Uploading Files

One of the more common use cases for web servers is to handle user uploads. In this example, you will simply save the uploaded file in a folder on the server; a more realistic example would likely push the uploaded file to a cloud-based storage system.

Exercise 13b.5 — Setting Up Node Project

1. Use the following command to install an additional package to handle the file uploading:

```
npm install multer
```

2. Briefly examine the documentation for the multer package by visiting the following:

`https://www.npmjs.com/package/multer`

3. Examine `public/uploader.html`. Modify the `<form>` element as follows:

```
<form method="post" enctype="multipart/form-data"
      action="/uploader" >
```

4. Add a following name to file <input> element.

```
<input type="file" name="fileElem" class="button">
```

5. Create a new folder named `uploads` in the root folder (i.e., same level as `public`, `scripts`, and `data`).

This folder will contain the files received from the user.

6. In the `scripts` folder, add a new file named `uploader.js` and add the following content.

```
const multer = require('multer');

// Set up storage specification for uploaded files
const storageSpec = multer.diskStorage({
  // indicate location for uploaded files
  destination: (req, file, cb) => {
      cb(null, 'uploads/');
  },
  // add the date+time to the filename of the uploaded file,
  // so each uploaded file has a unique filename on the server
  filename: (req, file, cb) => {
     cb(null, Date.now() + '-' + file.originalname);
  }
});

// Create the multer instance
const upload = multer({ storage: storageSpec });
module.exports = upload;
```

We are using the multer package to handle the uploaded file. We could have created all this functionality ourselves from scratch, but one of the significant benefits of the Node is the rich ecosystem of available packages.

7 In `stocks-api.js`, add the following:

```
app.use('/static', express.static(path.join(__dirname,'public')));
// Require the upload middleware
const upload = require('./scripts/uploader.js');
// Set up a route for file uploads
app.post('/uploader', upload.single('fileElem'), (req, res) => {
  // Handle the uploaded file
  res.json({ message: 'File uploaded successfully!' });
});
```

8 Ensure your stocks api is currently running by running the following command.

```
node --env-file=config.env stocks-api
```

9 Test by requesting `static/uploader.html` in the browser. Select a file and then submit. This should upload the file with a modified filename in your `uploads` folder.

Working with Web Sockets

One of the key benefits of the Node.js environment is its ability to create push-based applications. This ability makes use of WebSockets, an API that makes it possible to open an interactive (two-way) communication channel between the browser and a server outside of HTTP.

There are several WebSocket packages available via `npm`. In the following example, we will use Socket.io (http://socket.io/). Our example will consist of two files: the Node.js server application (`stocks-api.js`) that will receive and then push out received messages and the client file (`chat-client.html`) that will contain the user interface that sends and receives the chat messages.

Similarly Socket.io contains two JavaScript APIs: One that runs on the browser and one that runs on the server.

Exercise 13a.6 — Starting a Chat Application

1 Enter the following command:

```
npm install socket.io
```

This downloads the socket.io package and, thanks to the –save flag, adds a dependency to your package.json file.

2 Examine the `node_module` folder: notice that it contains several socket.io folders, which contains code for both the server and the client.

3 Examine `chat-client.html` (it's in the `public` folder) in the browser and then in an editor. Add the following reference to the `<head>`:

```
<script src="/socket.io/socket.io.js"></script>
```

You will now need to add a new handler for static requests to `'/socket.io'` below.

4 Add the following to `stocks-api.js`:

```
app.use('/static', express.static(path.join(__dirname, 'public')));
// map client request for socket.io file
app.use('/socket.io', express.static(path.join(__dirname,
    '/node_modules/socket.io/client-dist/')));
```

This will map requests for files within `'/socket.io'` to the appropriate folder within `node_module`. New versions of the socket.io package (installed in step 1) often changes its folder configuration; in such a case you will need to alter the path in this step.

Your client application might be a separate from your server application. In such a case, this step would be unnecessary. You would instead install socket.io-client in your client application folder structure.

5 Add the following to `stocks-api.js`:

```
// listen for socket communication on port 3000
const io = require('socket.io')(3000, {
    cors: {
        origin: ['http://localhost:8080']
    }
});
```

The latest version of socket.io now requires you to add in accepted origins for client requests. Since your client is running on localhost:8080, you have to add it as shown above. If you don't do this, you will see CORS errors in the browser console.

6 Also add the following to `stocks-api.js`:

```
io.on('connection', socket => {
    console.log('new connection made with client='+socket.id);
});
```

7 Add the following JavaScript to the supplied <script> element at the end of `chat-client.html`

```
<script>
  // chat will be on port 3000
  const socket = io('http://localhost:3000');
</script>
```

8 Test `stocks-api.js` and then request:

`http://localhost:8080/static/chat-client.html`

The Node console should display the new connection message.

9 Create a new browser tab (or switch to a different browser) and make the same request of `chat-client.html`.

Each separate request will trigger a new connection event. Your node console should display the message added in step 6 each time you add a new browser tab with the client.

Exercise 13a.7 — Adding Events to the Chat Application

1 Add the following to `stocks-api.js`:

```
io.on('connection', socket => {
    console.log('new connection made with client');

    // client has sent a new user has joined message
    socket.on('username', msg => {
        console.log('username: ' + msg);
        // attach passed username with this communication socket
        socket.username = msg;
        // broadcast message to all connected clients
        const obj = { message: "Has joined", user: msg };
        io.emit('user joined', obj);
    });
});
```

2 Add the following to `chat-client.html`:

```
const socket = io('http://localhost:3000');

// get user name, display it, and then tell the server
let username = prompt("What's your username?");
document.querySelector('.panel-header h3').textContent =
    'Chat [' + username + ']';
// send message to server
socket.emit('username', username);
```

3 Add the following to `chat-client.html`:

```
// a new user connection message has been received
socket.on('user joined', msg => {
  const li = document.createElement('li');
  li.innerHTML = `<em>${msg.user} - ${msg.message}</em>`;
  document.querySelector('#messages').appendChild(li);
});
```

4 Test `stocks-api.js` and then request the client (url is below) in two browser tabs:

`http://localhost:8080/static/chat-client.html`

The Node console should display the user name each time the client is requested the first time.

5 Add the following to `chat-client.html`:

```
// user has entered a new message
document.querySelector("#chatForm").addEventListener('submit', e
=> {
  e.preventDefault();
  const entry = document.querySelector("#entry");
  // send message to server
  socket.emit('chat from client', entry.value);
  entry.value = "";
});

// a new chat message has been received from server
socket.on('chat from server', msg => {
  const li = document.createElement('li');
  li.textContent = msg.user + ': ' + msg.message;
  document.querySelector('#messages').appendChild(li);
});
```

This code adds event handlers to implement the chat functionality.

7 In the connection handler, add the following to `stocks-api.js`:

```
io.on('connection', socket => {
   console.log('new connection made with client='+socket.id);

   // client has sent a new user has joined message
   socket.on('username', msg => {
      ...
   }

   // client has sent a chat message ... broadcast it
   socket.on('chat from client', msg => {
      console.log('message received from ' + socket.username);
      io.emit('chat from server',
                { user: socket.username, message: msg } );
   });
}
```

8 Test by requesting `chat-client.html` in several browser tabs.

9 Change the following line in `stocks-api.js` and test.

```
socket.broadcast.emit('user joined', obj);
```

This will broadcast this messages to every client except the one that generated it. The finished result should be similar to that in Figure 13-3.

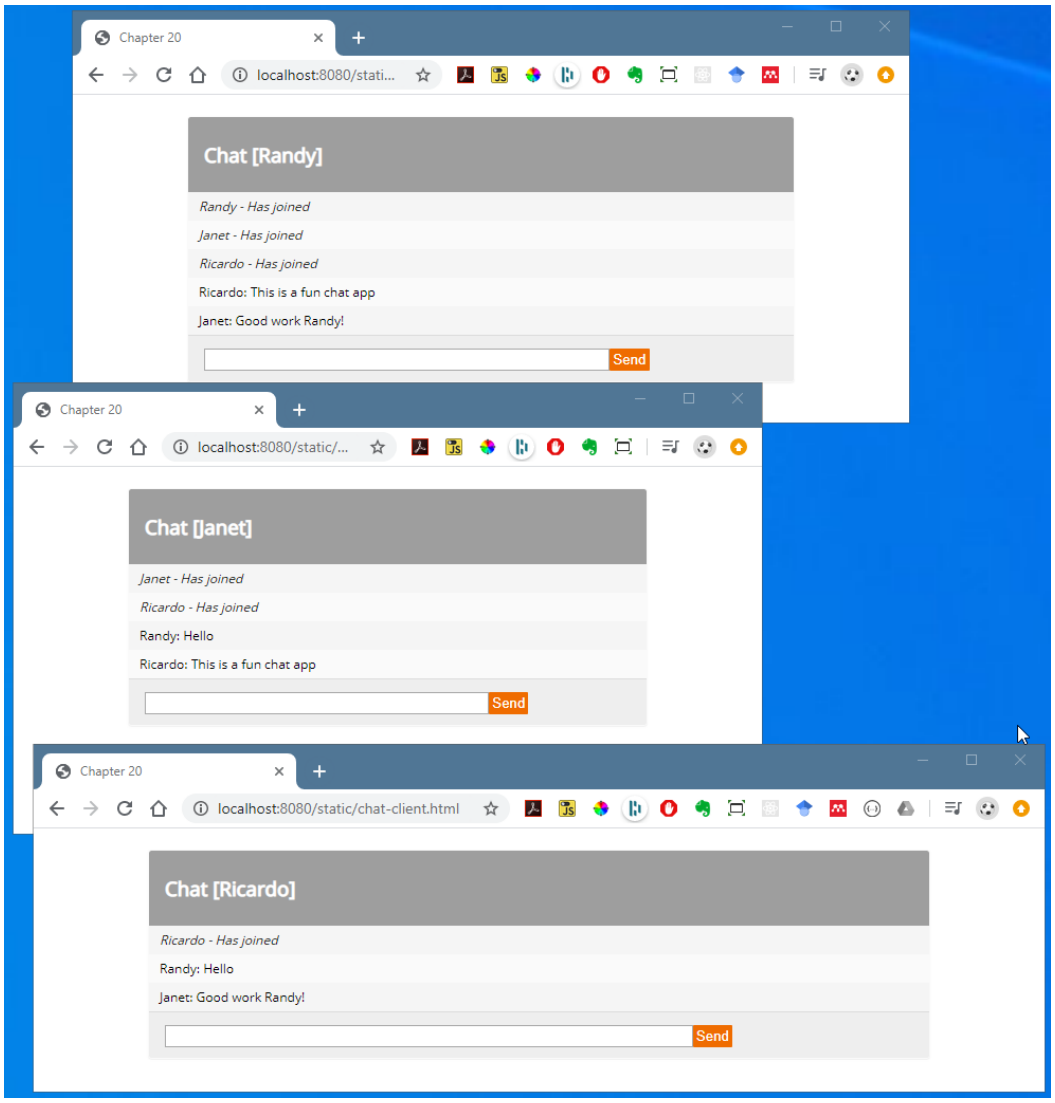

Figure 13.2 – Completed Chat Example