

LAB 13a

BEGINNING NODE

What You Will Learn

- How to install and use Node and npm
- How to create a static file server in Node
- How to use Express to simplify the process of writing applications in Node
- How to create an API that implements CRUD functionality
- 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 120 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

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

Date Last Revised: March 7, 2023

CREATING NODE APPLICATIONS

In this lab, you will be focusing on the server-side development environment Node.js (or Node for short). Like with PHP, you can work with it locally on your development machine or remotely on a server.

PREPARING DIRECTORIES

- 1 The starting `lab13a` folder has been provided for you (within the zip folder downloaded from Gumroad).

Exercise 13a.1 — USING NODE

- 1 The mechanisms for installing Node vary based on the operating system.

If you wish to run Node locally on a Windows-based development machine, you will need to download and run the Windows installer from the Node.js website.

If you want to run Node locally on a Mac, then you will have to download and run the install package.

If you want to run Node on a Linux-based environment, you will likely have to run `curl` and `sudo` commands to do so. The Node website provides instructions for most Linux environments.

If you are using a cloud-based development environment, Node is likely already installed in your workspace. If not, follow the instructions for that environment.
- 2 To run Node, you will need to use Terminal/Bash/Command Window. Verify it is working by typing the following commands:

```
node -v
npm -v
npx -v
```

The second command will display the version number of npm, the Node Package Manager which is part of the Node install. The third command (npx) is newer and might not be on your system: it is a tool for executing Node packages (though you can do so also via npm).

- 3 Navigate to the folder you are going to use for your source files in this lab.
- 4 Create a simple file from the command line via the following command:


```
echo "console.log('hello world')" > hello.js
```
- 5 Verify your node install works by running this file in node via the following command:

```
node hello.js
```

Notice that console.log in Node outputs to the terminal console, not to the browser console!

Exercise 13a.2 — SIMPLE SERVER

- 1 Edit (or create if you don't have it already on your development environment) the file `hello.js`.

- 2 Add (i.e., replace any existing code with) the following code and save:

```
/* Node applications make frequent use of node modules.
   A node module is simply a JS function library with
   some additional code that wraps the functions
   within an object.

   You can then make use of a module via the require()
   function. Most node applications make use of the
   very rich infrastructure of pre-existing modules
   available from npmjs.com

   The http module can be used to create an HTTP server
   */
const http = require('http');

// Configure HTTP server to respond with simple message to all requests
const server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello to our first node.js application\n");
  response.end();
});

// Listen on port 8080 on localhost
const port = 8080;
server.listen(port);

// display a message on the terminal
console.log("Server running at port=" + port);
```

- 3 Run the following command:

```
node hello.js
```

This executes the file in Node. You will see a message about the "Server running at port=8080" but nothing else. This application is a simple web server. That is, it is waiting for HTTP requests on port 8080. So you will need to make some requests using a browser.

- 4 In a browser, request this page. How you do so will vary depending on the environment you are using. If running Node locally on your machine, then you might simply need to request `http://localhost:8080`. If using a server-based Node environment, then you will have to request using the appropriate server URL.

If everything worked, you should see the Hello message in the browser window. This Node server will continue to run until you stop the application.

- 5 Try modifying the URL path in the browser by changing the url to `http://localhost:8080/path/products.html`.

It should make no difference to what the server does (that is, it ignores the path and/or query strings of the request and just returns the hello message for all requests).

- 6 Use Ctrl-C in the terminal to stop the hello server.

Anytime you want to modify and test Node file, you will have to stop the application (if running) and re-run it. Sometimes you have to press Ctrl-C repeatedly!

- 7 Try re-requesting `http://localhost/` in the browser.

It should display nothing (or some type of error message) since our `hello.js` file is no longer executing.

Our previous exercise created a rather one-dimensional server: all it did was display a hello message. In the next example, you will create a simple static web server that can serve HTML, SVG, PNG, and JSON files.

Exercise 13a.3 — FILE SERVER

- 1 If you haven't already done so, upload or copy the folder named `public` to your development location.
- 2 Create a new file named `file-server.js` (not in the `public` folder but one level above it).
- 3 Add the following code to this new file:

```
/* These additional modules allow us to process URL
   paths as well as read/write files; we will use these later */
const http = require("http");
const url = require("url");
const path = require("path");
const fs = require("fs");
```

- 4 Keep expanding this file by adding in the following code:

```
// begin by creating a HTTP server
const server = http.createServer(function (request, response) {

  // local folder path in front of the filename
  const filename = "public/sample.html";
  // read the file
  fs.readFile(filename, (err, file) => {
    // remember this is in callback function; it only gets
    // invoked once the file has been read in
    if (err) {
      response.writeHead(500, {"Content-Type": "text/html"});
      response.write(
        "<h1>500 Error - File not found</h1>\n");
    } else {
      response.writeHead(200, {"Content-Type": "text/html"});
      response.write(file);
    }
    response.end();
  });
});
```

- 5 Add the following code at the end of the document.

```
// Listen on port on localhost
let port = 8080;
server.listen(port);
// display a message on the terminal
console.log("Server running at port= " + port);
```

- 6 Save and test by running the command `node file-server.js` in the terminal. You may need to first exit the execution of any current node program by pressing `ctrl-c` in the terminal. You will also need to make a request (`http://localhost:8080/`) in the browser. The result will look similar to that shown in Figure 13.1.

From now on, when the lab says “test”, it means halting execution of current script (ctrl-c), running the required `node xxxs.js` command, and then making the appropriate request in the browser. At any rate, you should now see the content of `sample.html` in the browser.

- 7 Edit the filename as follows:

```
const filename = "public/venice.jpg";
```

- 8 Change the mime type as follows and test:

```
response.writeHead(200, { "Content-Type": "image/jpeg" });
response.write(file);
```

It will now display an image.

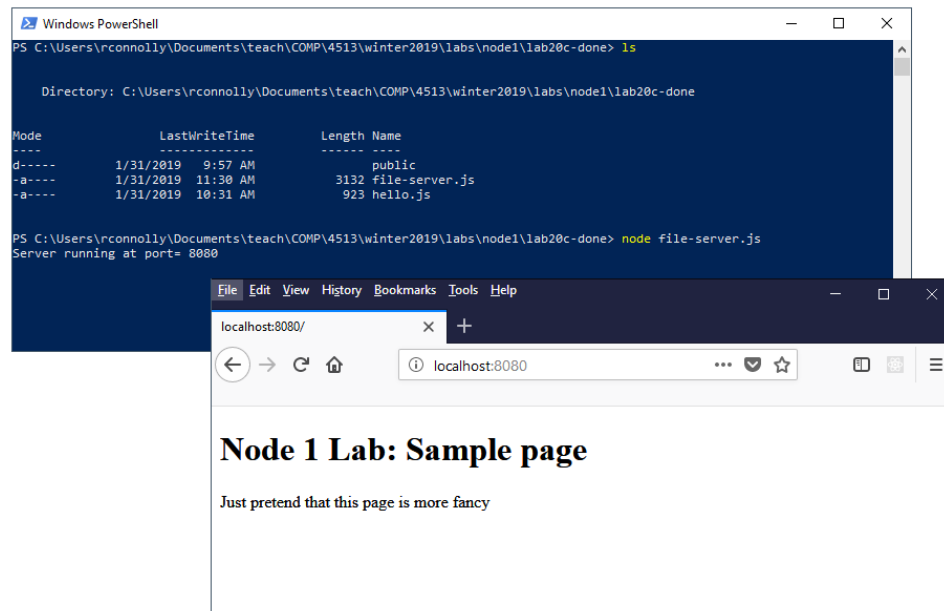


Figure 13.1 – Running the simple Node file server

Exercise 13a.4 — EXPANDING THE FILE SERVER

- 1 Keep expanding the `file-server.js` file by adding the following helper functions after the `require` lines at the top of the page.

```
// handler for errors
const output500Error = (response) => {
  response.writeHead(500, {"Content-Type": "text/html"});
  response.write("<h1>500 Error</h1>\n");
  response.write("Something went wrong with request\n");
  response.end();
};
```

- 2 Define the following array just after the `require` lines:

```
// maps file extention to MIME types
const mimeType = {
  '.html': 'text/html',
  '.json': 'application/json',
  '.jpg': 'image/jpeg',
  '.svg': 'image/svg+xml'
};
```

- 3 Comment out the code from step 4 in the previous exercise (Exercise 13a.3).

- 4 Now add the following code to create the server then save.

```
// our HTTP server now returns requested files
const server = http.createServer( (request, response) => {

  // get the filename from the URL
  let requestedFile = url.parse(request.url).pathname;
  // if no file provided in request, default to index.html
  if (requestedFile.length == 1) requestedFile = "/index.html";

  // now turn that into a file system file name by adding the
  // current local folder path in front of the filename
  const ourPath = process.cwd() + "/public";
  let filename = path.join(ourPath, requestedFile);
  console.log(filename);

  // try reading the file
  fs.readFile(filename, "binary", (err, file) => {
    if (err) {
      output500Error(response);
      return;
    }
    // based on the URL path, extract the file extension
    const ext = path.parse(filename).ext;

    // specify the mime type of file via header
    const header = { "Content-type": mimeType[ext] ||
                     "text/plain" };
    response.writeHead(200, header);
    // output the content of file
    response.write(file, "binary");
    response.end();
  });
});
```

- 7 Stop previous node execution via ctrl-c in terminal, then type

```
node file-server.js
```

- 8 In a browser request `http://localhost:8080/`.

This should display the file index.html

- 9 In a browser request `http://localhost:8080/venice.jpg` then
`http://localhost:8080/stocks-simple.json` and then
`http://localhost:8080/tester.html`

The result should look similar to that shown in Figure 13.2.

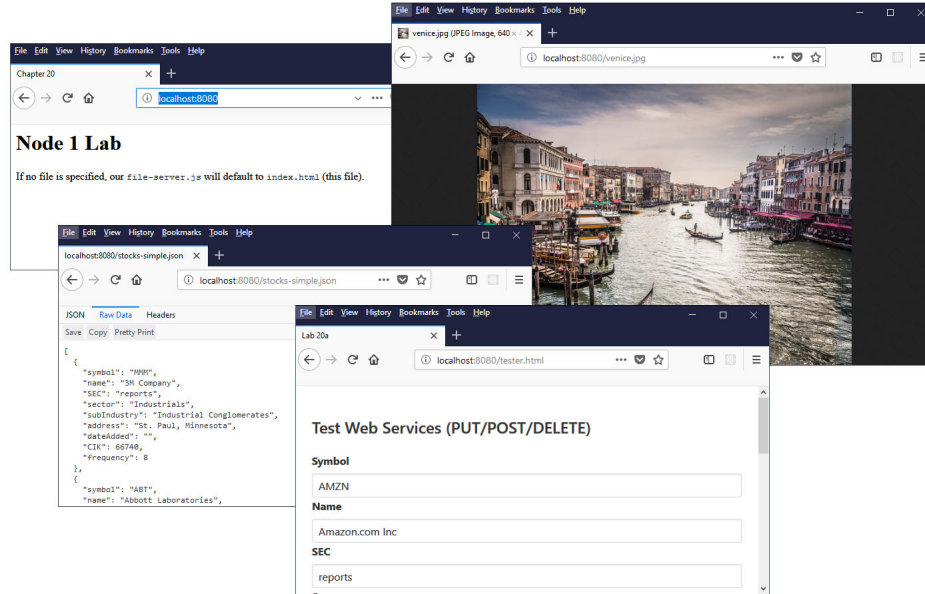


Figure 13.2 – Finished Node file server

USING EXPRESS

To reduce the amount of coding in Node, many developers make use of Express (or something similar), an external module that simplifies the development of server applications. In the next example, you will install and then use Express to develop a JSON-based web service. To do so, you will need to use npm, the Node Package Manager.

Exercise 13a.5 — USING NPM

- 1 In the terminal, type the following command:

```
npm init
```

This command will ask you a variety of questions and then create the package.json file, which is used to provide info about your application. You can also use this file to specify dependencies, that is, specify which modules (and their versions) your application uses.

- 2 For the different questions, use the following answers (blanks indicate blank or no answer):

```
sample-web-service
```

```
1.0.0
```

```
A sample web service to help learn Node
```

```
stocks-api.js
```

```
[you can skip through the next questions]
```

```
yes
```


3 Examine the `package.json` file that was created.

You can edit this file at any time.

4 Enter the following command:

```
npm install express
```

This downloads the express package (along with any packages it needs) and adds a dependency to your package.json file.

5 Examine your directory listing.

Notice that a new folder named node_modules has been created.

6 Examine the `node_modules` folder.

Installing express installed about 50 other modules. Every time you use the npm install command it adds the module files as a folder within node_modules.

7 Examine the `package.json` file.

Notice that a new dependency line has been added to the file.

8 Run the following command:

```
npm update
```

This command doesn't do anything right now. This command tells npm to see if there are new versions of any of the dependent modules, and if there is, download and install them.

9 Create a file named `.gitignore`, open it in an editor, and then add the following lines to it.

```
node_modules/  
.env
```

This tells git to not commit (i.e., ignore) the node_modules folder and the .env file (which is typically used to save environment information, which you almost never want saved in a repo). You could add other folders or file names to ignore if you so wished

Exercise 13a.6 — CREATING AN API

- 1 Create a new file named `stocks-simple.js`.

- 2 Add the following code to this new file:

```
// first reference required modules
const fs = require('fs');
const path = require('path');
const express = require('express');
```

- 3 Keep expanding this file by adding the following:

```
// for now, we will get our data by reading the provided json file
const jsonPath = path.join(__dirname, 'public',
                           'stocks-simple.json');
const jsonData = fs.readFileSync(jsonPath, 'utf8');
// convert string data into JSON object
const stocks = JSON.parse(jsonData);
```

```
// create an express app
const app = express();
```

This code reads in the JSON file (synchronously this time) and sets up express

- 4 Now add the remaining code to this same file:

```
// return all the stocks when a root request arrives
app.get('/', (req, resp) => { resp.json(stocks) } );

// Use express to listen to port
let port = 8080;
app.listen(port, () => {
  console.log("Server running at port= " + port);
});
```

- 5 Test by running (via entering `node stocks-simple` into the terminal) and viewing in browser (i.e., request `http://localhost:8080/`).

This should display the contents of the JSON file.

To make this web service more useful, you will need to add additional **routes**. In Express, routing refers to the process of determining how an application will respond to a request. For instance, instead of displaying all the stocks, we might only want to display a single stock identified by its symbol, or a subset of stocks based on a criteria. These different requests are typically distinguished via different URL paths (instead of using query string parameters).

In the next exercise, the following routes will be supported:

ROUTE	EXAMPLE	DESCRIPTION
/	<i>domain/</i>	Return JSON for all stocks
/stock/:symbol	<i>domain/stock/amzn</i>	Return JSON for single stock whose symbol is 'AMZN'
/stock/name/:substring	<i>domain /stock/name/alpha</i>	Return JSON for any stocks whose name contains the text 'alpha'

Exercise 13a.7 — ADDING ADDITIONAL ROUTING

- 1 Make a copy of `stocks-simple.js` and call it `stocks-api.js`.

- 2 Modify the following line as follows:

```
const jsonPath = path.join(__dirname, 'public',  
                             'stocks-complete.json');
```

- 3 Add the following route definition after your already existing one:

```
// return just the requested stock  
app.get('/stock/:symbol', (req,resp) => {  
  // change user supplied symbol to upper case  
  const symbolToFind = req.params.symbol.toUpperCase();  
  // search the array of objects for a match  
  const matches =  
    stocks.filter(obj => symbolToFind === obj.symbol);  
  // return the matching stock  
  resp.json(matches);  
});
```

- 4 Run the file (breaking then enter `node stocks-api`) and test in browser using for instance the following URL in the browser: `http://localhost:8080/stock/amzn`

- 5 Add the following additional route definition.

```
// return all the stocks whose name contains the supplied text
app.get('/stock/name/:substring', (req,resp) => {
  // change user supplied substring to lower case
  const substring = req.params.substring.toLowerCase();
  // search the array of objects for a match
  const matches = stocks.filter( (obj) =>
    obj.name.toLowerCase().includes(substring) );
  // return the matching stocks
  resp.json(matches);
});
```

- 6 Test (break+run) in browser using for instance the following URL in the browser:
http://localhost:8080/stock/name/alph

Exercise 13a.8 — ADDING STATIC FILE SERVING

- 1 Add an additional route as follows:

```
const app = express();
// handle requests for static resources
app.use('/static',
  express.static(path.join(__dirname, 'public')));
```

The first parameter specifies the “virtual” path that the outside world will use; the second parameter specifies the “actual” path on the server.

- 2 Test by making the following request:

```
http://localhost:8080/static/tester.html
http://localhost:8080/static/venice.jpg
```

Exercise 13a.9 — CREATING A MODULE

- 1 Make a copy of `stocks-api.js` and call it `stocks-api-backup.js`.

In this exercise, you will keep modifying `stocks-api.js` file. Doing this step provides you with a working backup of the previous exercise.

- 2 Create a new subfolder named `scripts`.
- 3 In that folder, create a new file named `data-provider.js`.
- 4 In this file add the following code:

```
const path = require("path");
const fs = require("fs");
```

- 5 From `stocks-api.js`, **cut** the following lines of code and paste it into our new file with a change to the folder path as shown below.

```
// for now, we will get our data by reading the provided json file
const jsonPath = path.join(__dirname, '../public',
                           'stocks-complete.json');
const jsonData = fs.readFileSync(jsonPath, 'utf8');
// convert string data into JSON object
const stocks = JSON.parse(jsonData);
```

- 6 Add the following line of code to the end of this file and save.

```
module.exports = stocks;
```

Anything you add to the `module.exports` property will become available to users of this module.

- 7 In `stocks-api.js`, add the following code that uses this new module.

```
const app = express();
// reference our own modules
const stocks = require('../scripts/data-provider.js');
```

- 8 Save both files and test (breaking then enter `node stocks-api` and test in browser using the following URL in the browser): `localhost:8080/stock/amzn`

Exercise 13a.10 — ENHANCING THE MODULE

- 1 In the `scripts` folder, create a new file named `stock-router.js`.

This example module will illustrate a different approach; rather than exporting a router, this module will export an object containing multiple methods.

- 2 In this file add the following structure.

```
/* Module for handling specific requests/routes for stock data */
// return all the stocks when a root request arrives
const handleAllStocks = (stocks, app) => {

};

// return just the requested stock
const handleSingleSymbol = (stocks, app) => {

};

// return all the stocks whose name contains the supplied text
const handleNameSearch = (stocks, app) => {

};

module.exports = {
  handleAllStocks,
  handleSingleSymbol,
  handleNameSearch
};
```

If a module needs to export multiple properties, simply define them in an object and set `module.exports` equal to that object.

- 3 **Move** the symbol route code from step 6 of Exercise 13a.6 of `stocks-api.js` into the new module so your code looks similar to the following:

```
// return all the stocks when a root request arrives
const handleAllStocks = (stocks, app) => {
  app.get('/', (req,resp) => { resp.json(stocks) } );
};
```

- 4 **Move** the symbol route code from step 3 of Exercise 13a.7 of `stocks-api.js` into the new module so your code looks similar to the following:

```
const handleSingleSymbol = (stocks, app) => {  
  app.get('/stock/:symbol', (req,resp) => {  
    // change user supplied symbol to upper case  
    const symbolToFind = req.params.symbol.toUpperCase();  
    // search the array of objects for a match  
    const stock = stocks.filter(s => symbolToFind ===  
                                   s.symbol);  
    // return the matching stock  
    resp.json(stock);  
  });  
};
```

- 5 Move the symbol route code from step 5 of Exercise 13a.7 of `stocks-api.js` into the new module so your code looks similar to the following:

```
const handleNameSearch = (stocks, app) => {  
  app.get('/stock/name/:substring', (req,resp) => {  
    // change user supplied substring to lower case  
    const substring = req.params.substring.toLowerCase();  
    // search the array of objects for a match  
    const matches = stocks.filter( s =>  
                                   s.name.toLowerCase().includes(substring) );  
    // return the matching stocks  
    resp.json(matches);  
  });  
}
```

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

```
// reference our own modules  
const stocks = require('./scripts/data-provider.js');  
const stockRouter = require('./scripts/stock-router.js');
```

- 7 In `stocks-api.js`, tell Express to use this router within `stock-router.js`:

```
app.use("/static", express.static(path.join(__dirname, "public")));  
// handle other requests for stocks  
stockRouter.handleSingleSymbol(stocks, app);  
stockRouter.handleNameSearch(stocks, app);
```

- 8 Since we have moved functionality into modules, you can remove the `require` statements for `fs` from `stocks-api.js`.

- 9 Test (break+run) using the following URLs in the browser:

```
http://localhost:8080/stock/name/alph
http://localhost:8080/stock/amzn
http://localhost:8080/stock/
http://localhost:8080/static/tester.html
```

By using modules, our stocks-api.js file is about 25 lines.

The code for the stock routes assumes that the requested symbol name or substring exists in our data file. This is certainly an unrealistic assumption. The next exercise adds some complexity to handle unexpected inputs.

Exercise 13a.11 — ADDING IN SOME ERROR HANDLING

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

```
// error messages need to be returned in JSON format
const jsonMessage = (msg) => {
  return { message : msg };
};
```

- 2 Edit `handleSingleSymbol()` as follows (some code omitted).

```
const handleSingleSymbol = (stocks, app) => {
  ...
  const matches = stocks.filter(obj =>
                                symbolToFind === obj.symbol);

  // return the matching stock
  if (matches.length > 0) {
    resp.json(matches);
  } else {
    resp.json(jsonMessage(`Symbol ${symbolToFind} not found`));
  }
  ...
}
```

- 3 Edit `handleNameSearch()` as follows (some code omitted).

```
const handleNameSearch = (stocks, app) => {
  ...
  const matches = stocks.filter( (obj) =>
                                obj.name.toLowerCase().includes(substring) );
  // return the matching stocks
  if (matches.length > 0) {
    resp.json(matches);
  } else {
    resp.json(jsonMessage(
      `No symbol matches found for ${substring}`));
  }
  ...
}
```


- 4 Test (break+run) using the following URLs in the browser:

```
http://localhost:8080/stock/name/ksdfskdfh  
http://localhost:8080/stock/amznGGG
```

These should display appropriate error messages, in JSON format, since there are no matching stocks for these search terms.

FETCHING EXTERNAL DATA

In one of your earlier JavaScript labs, you would have learned about the `fetch()` function, which allowed client-side JavaScript to asynchronously consume external APIs. At the time of writing Node does not support `fetch()`, so we will have to use an external module such as `https.get`, `axios`, `r2`, or `node-fetch`. In the next exercise, we will use `node-fetch` since its usage is essentially the same as `fetch()` on the client-side.

Exercise 13a.12 — CONSUMING AN API

- 1 In 2022, Node finally supported the Fetch API (though you may still get an ‘experimental feature’ warning). In `stocks-router.js`, add the following method.

```
async function retrievePriceData(symbol, resp) {  
  const url =  
    `http://www.randyconnolly.com/funwebdev/3rd/api/stocks/history.php?symbol=${symbol}`;  
  
  // retrieve the response then the json  
  const response = await fetch(url);  
  const prices = await response.json();  
  // return the retrieved price data  
  resp.json(prices);  
}
```

Notice that we are using the newer JavaScript `async...await` syntax to make our code cleaner. It is important to remember that the request for the external API is still happening asynchronously. You might be tempted to return prices from this function; it won't return with the data however, but only a (pending) Promise object.

- 2 In the same file, add the following method.

```
// return daily price data
const handlePriceData = (stocks, app) => {
  app.get('/stock/daily/:symbol', (req, resp) => {
    // change user supplied symbol to upper case
    const symbolToFind = req.params.symbol.toUpperCase();
    // search the array of objects for a match
    const stock = stocks.filter(s => symbolToFind === s.symbol);
    // now get the daily price data
    if (stock.length > 0) {
      retrievePriceData(symbolToFind, resp);
    } else {
      resp.json(jsonMessage(`Symbol ${symbolToFind} not
found`));
    }
  });
}
```

- 3 Add this new route to the exports as shown below.

```
module.exports = {
  handleAllStocks,
  handleSingleSymbol,
  handleNameSearch,
  handlePriceData
};
```

- 4 Now edit `stock-api.js` and add the new route handler as shown in the following.

```
// handle other requests for stocks
stockRouter.handleSingleSymbol(stocks, app);
stockRouter.handleNameSearch(stocks, app);
stockRouter.handlePriceData(stocks, app);
```

- 5 Test (break+run) using the following URLs in the browser:

```
http://localhost:8080/stock/daily/adbe
http://localhost:8080/stock/daily/jskdfh
```

The first should display about three months worth of sample data , while the second returns an error message.

It should be noted that typically in Node, you retrieve data for an API from some type of database (rather than from another API as in the previous exercise). This will be covered in Lab 14b.

Test Your Knowledge #1

- 1 Make a backup copy of `data-provider.js` and `stocks-api.js`. In this exercise, you will modify the original so that it instead fetches its data from the external API at:

`https://www.randyconnolly.com/funwebdev/3rd/api/stocks/companies.php`

This API should be available at http and at https.

- 2 Comment out the existing code in `data-provider.js` and replace with the following:

```
async function retrieveCompanies(app) {
  const url =
`https://www.randyconnolly.com/funwebdev/3rd/api/stocks/companies
.php`;

  // use fetch and await to get stocks data
  const response = await fetch(url);
  const stocks = await response.json();

  // handle other requests for stocks
  stockRouter.handleAllStocks(stocks, app);
  stockRouter.handleSingleSymbol(stocks, app);
  stockRouter.handleNameSearch(stocks, app);
  stockRouter.handlePriceData(stocks, app);
}

module.exports = { retrieveCompanies };
```

- 3 Change `stocks-api.js` as follows (some code will move to `data-provider.js`):

```
const companyProvider = require('./scripts/data-provider.js');
const stockRouter = require("./scripts/stock-router.js");
companyProvider.retrieveCompanies(app, stockRouter);

// handle requests for static resources
app.use("/static", express.static(path.join(__dirname, "public")));

// Use express to listen to port
let port = 8080;
...
```

- 4 Test with the following requests (which should all work).

```
http://localhost:8080/
http://localhost:8080/stock/adbe
http://localhost:8080/stock/name/alph
http://localhost:8080/static/tester.html
http://localhost:8080/stock/daily/adbe
```

Notice that your stock API now provides additional data.

IMPLEMENTING CRUD BEHAVIORS

For JavaScript intensive applications, it is common for web APIs to provide not only the ability to retrieve data, 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.

In the next set of exercises, you will add this CRUD functionality. A form has been provided that makes the `POST/PUT/DELETE` requests. In this example, your code will simply modify the in-memory JSON array. In a future lab, you will make such data changes persistent using a database.

Exercise 13a.13 — ADDING UPDATE SUPPORT

- 1 Examine `tester.html` and `tester.js` in the `public` folder.
This form will allow you to test the CRUD functionality of your web service.
- 2 Make a copy of `stock-router.js` and call it `stock-router-backup.js`.
- 3 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 provide us with access to form querystring data.

- 4 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) => {
    ...
  });

  // if it is a PUT request then update specified stock
  app.put('/stock/:symbol', (req,resp) => {
    const symbolToUpd = req.params.symbol.toUpperCase();
    // use lodash to 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`));
    }
  });
};
```

- 5 Save and test by requesting the `static/tester.html` file. It already has a prefilled in form. Edit some of the fields (but **not** the symbol field) and click the **Update** button.
- 6 After updating, in a separate tab, make a GET request for the same symbol using:

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

It should contain the updated values.

- 7 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 Lab13b, you will make changes persistent by recording them in a database.

Right now, your `stock-router.js` file is becoming quite large (and you still need to add two more CRUD behaviors). Further, it has become quite incohesive, in that it contains routing logic as well as implementing the behaviors for different routing requests. In the next exercise, you will move the behaviors into a separate module.

Exercise 13a.14 — REFACTORING THE STOCK ROUTER

- 1 In the `scripts` folder, create a new file named `stockController.js`.
- 2 In this new file, **move** the `jsonMessage()` function and the `retrievePriceData()` functions from `stock-router.js` to this new file.

- 3 In this new file, add the following code.

```
const findSymbol = (stocks, req, resp) => {
};

const updateSymbol = (stocks, req, resp) => {
};

const findName = (stocks, req, resp) => {
};

const findPrices = (stocks, req, resp) => {
};

module.exports = {
  findSymbol,
  updateSymbol,
  findName,
  findPrices
};
```

- 4 **Move** the following GET content from `handlePriceData()` in `stock-router.js` to `findPrices()` in `stockController.js`.

```
const findPrices = (stocks, req, resp) => {
  const symbolToFind = req.params.symbol.toUpperCase();
  // search the array of objects for a match
  const matches = stocks.filter(s => symbolToFind === s.symbol);
  if (matches.length > 0) {
    // now get the hourly price data from IEX
    retrievePriceData(symbolToFind, resp);
  } else {
    resp.json(jsonMessage(`Symbol ${symbolToFind} not found`));
  }
};
```

- 5 Move the following content from `handleNameSearch()` in `stock-router.js` to `findName()` in `stockController.js`.

```
const findName = (stocks, req, resp) => {
  const substring = req.params.substring.toLowerCase();
  // search the array of objects for a match
  const matches = stocks.filter( s =>
    s.name.toLowerCase().includes(substring) );
  if (matches.length > 0) {
    // return the matching stocks
    resp.json(matches);
  } else {
    resp.json(jsonMessage(
      `No symbol matches found for ${substring}`));
  }
};
```

- 6 Move the following GET content from `handleSingleSymbol()` in `stock-router.js` to `findSymbol()` in `stockController.js`.

```
const findSymbol = (stocks, req, resp) => {
  const symbolToFind = req.params.symbol.toUpperCase();
  // search the array of objects for a match
  const matches = stocks.filter(s => symbolToFind === s.symbol);
  // return the matching stock
  if (matches.length > 0) {
    resp.json(stock);
  } else {
    resp.json(jsonMessage(`Symbol ${symbolToFind} not found`));
  }
};
```

- 7 Move the following PUT content from `handleSingleSymbol()` in `stock-router.js` to `updateSymbol()` in `stockController.js`.

```
const updateSymbol = (stocks, req, resp) => {
  const symbolToUpd = req.params.symbol.toUpperCase();
  // use lodash module to 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(`${symbolToUpd} not found`));
  } else {
    // symbol found in our stock array, so replace its value
    // with those from form
    stocks[indx] = req.body;
    // let requestor know it worked
    resp.json(jsonMessage(`${symbolToUpd} updated`));
  }
};
```

- 8 In `stock-router.js`, replace the moved code with invocations to the appropriate `stockController` methods. The entire `stock-router.js` content should look like the following (some code omitted).

```
const stockController = require('./stockController.js');

// return just the requested stock
const handleSingleSymbol = (stocks, app) => {
  app.get('/stock/:symbol', (req,resp) => {
    stockController.findSymbol(stocks,req,resp);
  });
  // if it is a PUT request then update specified stock
  app.put('/stock/:symbol', (req,resp) => {
    stockController.updateSymbol(stocks,req,resp);
  });
};

// return all the stocks whose name contains the supplied text
const handleNameSearch = (stocks, app) => {
  app.get('/stock/name/:substring', (req,resp) => {
    stockController.findName(stocks,req,resp);
  });
};

// return daily price data
const handlePriceData = (stocks, app) => {
  app.get('/stock/daily/:symbol', (req,resp) => {
    stockController.findPrices(stocks,req,resp);
  });
}
module.exports = { ... };
```

- 9 Test using the following urls:

```
http://localhost:8080/stock/ADBE
http://localhost:8080/stock/name/alp
http://localhost:8080/stock/daily/amzn
http://localhost:8080/static/tester.html
```

While we haven't added any functionality, we have made our code base simpler, in that `stock-router.js` only processes the routes, and lets `stockController.js` implement the behaviors for the routes.

- 10 Modify `handleSingleSymbol()` as follows and test (it doesn't change the functionality it just changes the style of the code).

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

    .put( (req,resp) => {
      stockController.updateSymbol(stocks,req,resp);
    });
};
```

Test Your Knowledge #2

In this Test Your Knowledge, you will complete the remaining CRUD functionality in `stock-api.js`.

- 1 Add the following to the `handleSingleSymbol()` function in `stock-router.js`.

```
const handleSingleSymbol = (stocks, app) => {
  app.route('/stock/:symbol')
    // if it is a GET request then return specified stock
    .get( (req,resp) => {
      stockController.findSymbol(stocks,req,resp);
    })
    // if it is a PUT request then update specified stock
    .put( (req,resp) => {
      stockController.updateSymbol(stocks,req,resp);
    })
    // if it is a POST request then insert new stock
    .post( (req,resp) => {
      stockController.insertSymbol(stocks,req,resp);
    })
    // if it is a DELETE request then delete specified stock
    .delete( (req,resp) => {
      stockController.deleteSymbol(stocks,req,resp);
    });
};
```

- 2 Implement `insertSymbol()` in `stockController.js`. To add a new symbol, simply push a new stock object (populated from the `req.body` parameter) onto the `stocks` array.
- 3 Implement `deleteSymbol()` in `stockController.js`. To remove a stock from the `stocks` array, use the `slice()` function.
- 4 Remember to add `insertSymbol` and `deleteSymbol` to the `module.exports`.

5 Test using:

```
http://localhost:8080/static/tester.html
http://localhost:8080/stock/ADBE
```

Easiest way to test is to first use Delete button on ADBE, then use stock API and request ADBE (which should display error message). Then use Insert button on ADBE, then use stock API and request ADBE (which will now work, but will only contain the form data).

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.15 — 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.16 — 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 = `${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.

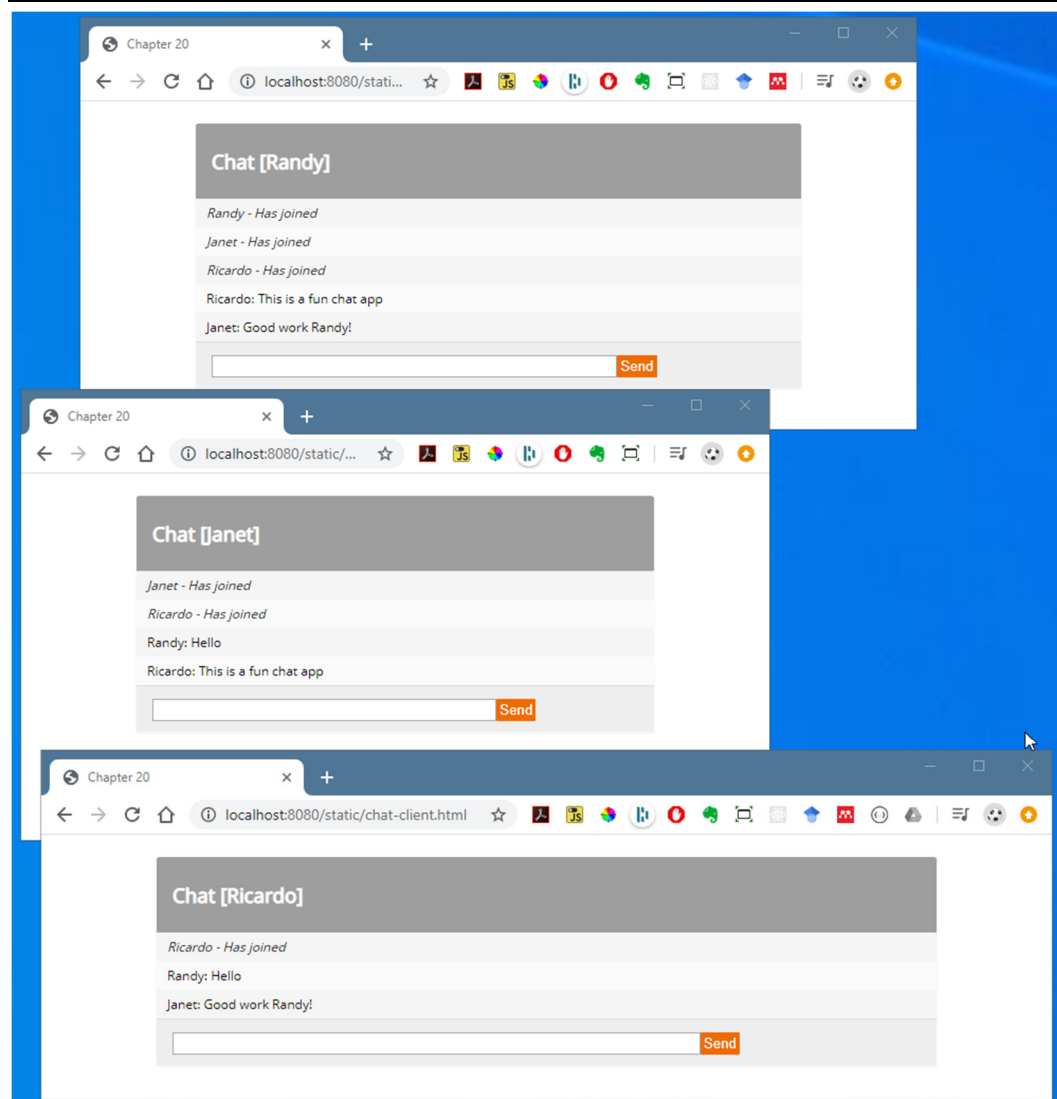


Figure 13.3 – Completed Chat Example