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

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: Jan 13, 2024
Revisions: Split previous content into Lab13a+Lab13b; added asynchronous; improved instructions; env file

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. **You can also use the Terminal directly within Visual Code.**

   Verify Node 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
   ```

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

Exercise 13a.2 — Simple Server

1 Create a new file `simple-server.js` with the following code and save.

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

// Create HTTP server to respond with simple message to all requests
const server = http.createServer( (request, response) => {
   console.log("request from " + request.url);
   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);
```

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.

2 Run the following command:

```
node simple-server
```

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. Thus you will need to make some requests using a browser.

3 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 message in the browser window. This Node server will continue to run until you stop the application.

4 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.

5 Use Ctrl-C in the terminal to stop the simple 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!

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

It should display nothing (or some type of error message) since your `simple-server.js` file is no longer executing.

7 Edit the code as follows.

```
const server = http.createServer( (request, response) => {
   console.log("request from " + request.url);
   response.writeHead(200, {"Content-Type": "text/html"});
   response.write("<html><body>Hello again, this time as ");
   response.write("<strong>HTML</strong></body></html>");
   response.end();
});
```

8 Test by running `node server` and then making request in browser.

9 Stop execution (via Ctrl-c), make the following change, and try to test.

```
const server = http.createAAAAServer( (request, response) => {
```

This will generate a compile error. Node error messages can be quite verbose since they include a stack trace.

10 Fix the error and retest.

From now on, when the lab says "test", it means halting execution of current script (ctrl-c), running the required `node ????` command, and then making the appropriate request in the browser.

11 Add the following code and test.

```
console.log("dirname = " + __dirname);
console.log("filename = " + __filename);
```

This code references two of Node's global variables. They are available everywhere in Node.

The previous exercise created a rather one-dimensional server: all it did was display a single 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 Create a new file named `file-server.js`.

2 Add the following code to this new file:

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

3 Keep expanding this file by adding in the following code:

```
const server = http.createServer( (req, resp) => {
   // local folder path in front of the filename
   const filename = "public/sample.html";
   // read the file asynchronously
   fs.readFile(filename, (err, file) => {
      // remember this is in callback function; it only gets
      // invoked once the file has been read in
      if (err) {
         resp.writeHead(500, {"Content-Type": "text/html"});
         resp.write(
             "<h1>500 Error - File not found</h1>\n");
      } else {
         resp.writeHead(200, {"Content-Type": "text/html"});
         resp.write(file);
      }
      resp.end();
   });
});
let port = 8080;
server.listen(port);
console.log("Server running at port= " + port);
```

4 Save and test. You may also want to examine the file `sample.html` in the `public` folder.

The result will look similar to that shown in Figure 13a.1.

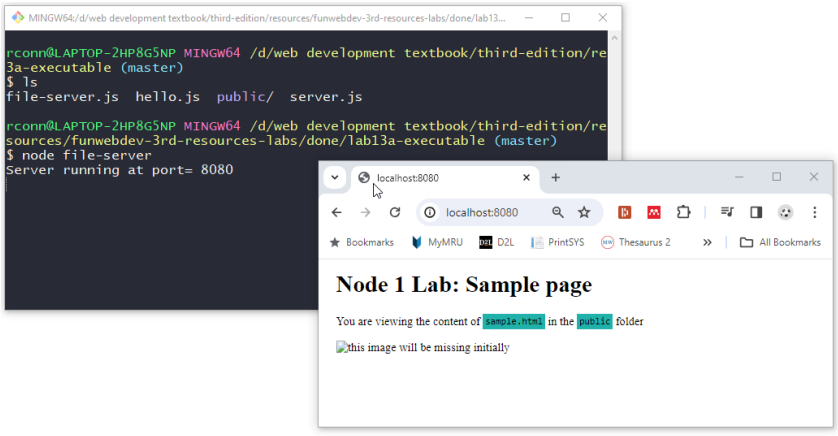

Figure 13a.1 – Running the simple Node file server

7 Edit the filename as follows and test.

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

This should display the file not found error. In your browser, you can examine the received status codes and headers via the Network tab in Chrome or Firefox, as shown in Figure 13a.2.

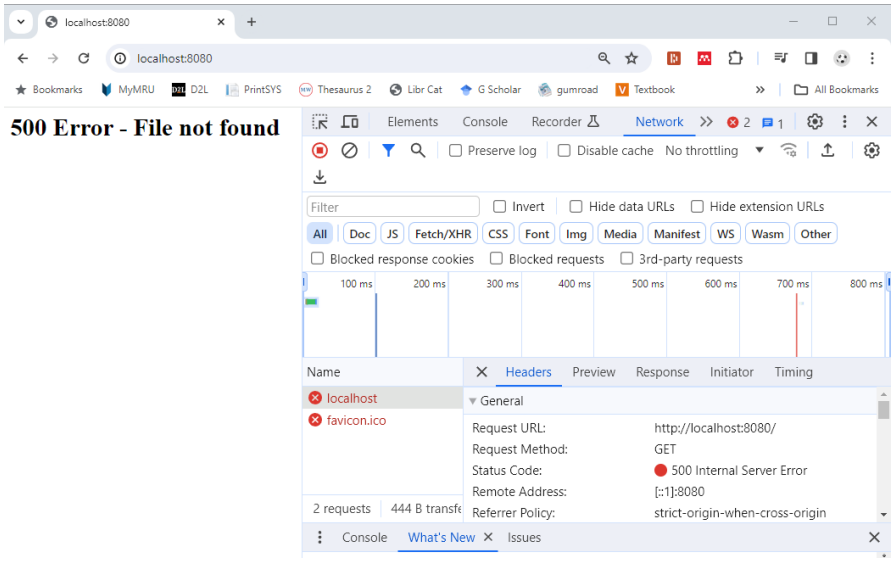

Figure 13a.2 – Examining the response

8 Edit the filename as follows and test.

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

This will display a bunch of seemingly random nonsense characters. The problem is that our server sent the contents of the venice.jpg image file but the browser thinks it received HTML content.

9 Change the mime type as follows and test:

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

It will now display an image.

10 Delete or comment out the `readFile` code from step 3 and replace it with the following.

```
const server = http.createServer( (req, resp) => {
   // local folder path in front of the filename
   const filename = "public/sample.html";
   // read the file synchronously
   try {
     const file = fs.readFileSync(filename);
     resp.writeHead(200, {"Content-Type": "image/jpeg"});
     resp.write(file);
     resp.end();
   } catch {
     resp.writeHead(500, {"Content-Type": "text/html"});
     resp.write(
            "<h1>500 Error - File not found</h1>\n");
     resp.end();
   }
});
```

While this creates somewhat simpler code, in general we want to avoid blocking/synchronous code in Node

Exercise 13a.4 — Expanding the File Server

1 Make a copy of `file-server.js` and name it `file-server2.js`.

2 Add the following to `file-server2.js` file by adding the following after the `require` lines at the top of the page.

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

// 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();
};

// maps file extention to MIME types
const mimeTypes = [
   ['.html', 'text/html'],
   ['.json', 'application/json'],
   ['.jpg', 'image/jpeg'],
   ['.svg', 'image/svg+xml']
];
```

3 Now replace the code for the `createServer` function as follows.

```
const server = http.createServer( (req, resp) => {
   // get the filename from the URL
  let urlFile = url.parse(req.url).pathname;
   // if no file provided in request, default to index.html
  if (urlFile.length == 1) urlFile = "/index.html";
  console.log("Filename in URL=" + urlFile);

  // turn it into the actual file system filename
  const localPath = __dirname + "/public";
  let localFile = path.join(localPath, urlFile);
  console.log("Filename on device=" + localFile);

  // try reading the file
  fs.readFile(localFile, (err, file) => {
    if (err) {
      output500Error(response);
      return;
    }
    // based on the URL path, extract the file extension
    const ext = path.parse(filename).ext;
    // lookup mime type for this extension
    const mime = mimeTypes.find( m => m[0] == ext);

    // specify the mime type of file via header
    const header = { "Content-type": mime[1] ||
                          "text/plain" };
    resp.writeHead(200, header);
    // output the content of file
    resp.write(file);
    resp.end();
  });
});
```

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

```
node file-server2
```

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/simple.json` and then `http://localhost:8080/tester.html`

The result should look similar to that shown in Figure 13a.3.

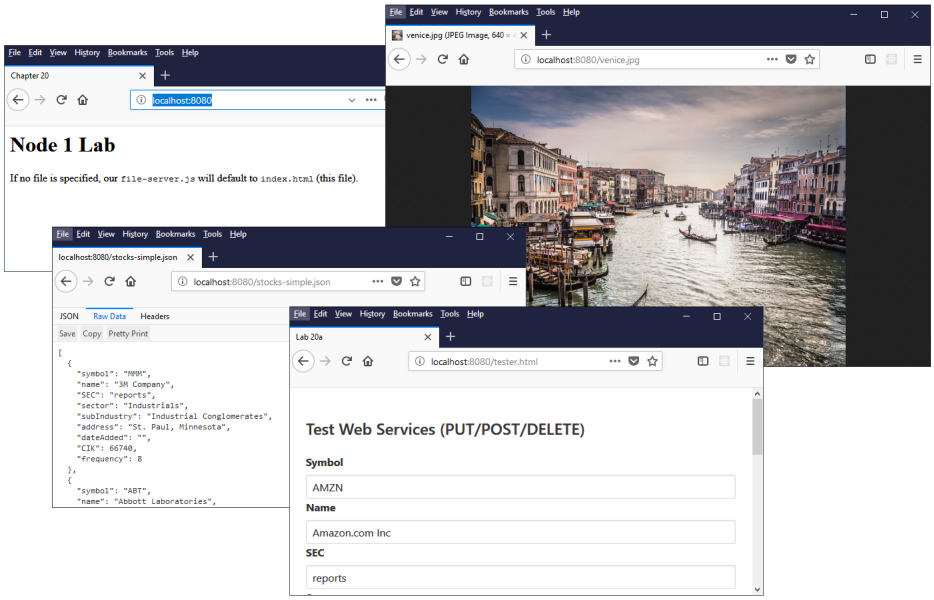

Figure 13a.3 – Finished Node file server

Exercise 13a.5 — Asynchronous Patterns Using Promises

1. Make a copy of `file-server2.js` and name it `file-server3promise.js`.

2. Modify the following `require` at the top of the page.

```
const fsProm = require("fs").promises;
```

3. Modify the `readFile` invocation as follows.

```
// read the file
fsProm.readFile(localFile)
   .then( contents => {
      // based on the URL path, extract the file extension
      const ext = path.parse(localFile).ext;
      // lookup mime type for this extension
      const mime = mimeTypes.find( m => m[0] == ext);

      // specify the mime type of file via header
      const header = { "Content-type": mime[1] || "text/plain" };
      resp.writeHead(200, header);
      // output the content of file
      resp.write(contents);
      resp.end();
   })
   .catch( err => {
      output500Error(resp);
   });
```

Compare this version to the one in Exercise 13a.4. Using promises for asynchronous coding results in cleaner code than with callback functions. NOTE: not all asynchronous libraries support patterns!

4 Test by requesting `http://localhost:8080/tester.html`.

5 Make a copy of `file-server3promise.js` and name it `file-server4async.js`.

6 Modify it as follows (some code omitted).

```
const server = http.createServer( async (req, resp) => {
   ...
   // read the file
  try {
    const contents = await fsProm.readFile(localFile);
    // based on the URL path, extract the file extension
    const ext = path.parse(localFile).ext;
    // lookup mime type for this extension
    const mime = mimeTypes.find( m => m[0] == ext);

    // specify the mime type of file via header
    const header = { "Content-type": mime[1] || "text/plain" };
    resp.writeHead(200, header);
    // output the content of file
    resp.write(contents);
    resp.end();
  }
  catch {
    output500Error(resp);
  }
```

This is probably the most commonly used pattern in 2024 for asynchronous coding in Node for modules that support promises.

7 Test by requesting `http://localhost:8080/tester.html`.

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.6 — 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; you can also just leave them all blank as well, as it makes no difference here.

```
sample-api
1.0.0
A sample api 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
*.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.7 — Creating an API Using Express

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

2 Add the following code to this new file:

```
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 file = 'stocks-simple.json';
const jsonPath = path.join(__dirname, 'data', file);
// read file contents synchronously
const jsonData = fs.readFileSync(jsonPath, 'utf8');
// convert string data into JSON object
const stocks = JSON.parse(jsonData);

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

4 Now add the remaining code to this same file:

```
// define the API routes

// 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.

6 Test by making the following request: `http://localhost:8080/asd`

Express will send a 404 Not Found response, since this path (/asd) hasn't been defined in our stocks-simple code.

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
/	*domain*/	Return JSON for all stocks
/stock/:symbol	*domain*/stock/amzn	Return JSON for single stock whose symbol is 'AMZN'
/stock/name/:substring	*domain*/stock/name/alpha	Return JSON for any stocks whose name contains the text 'alpha'

Exercise 13a.8 — 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 file = '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 Test using 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 using the following URL in the browser:
`http://localhost:8080/stock/name/alph`

Exercise 13a.9 — Adding Static File Serving

1 Add a middleware function 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.10 — 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 file = 'stocks-complete.json';
const jsonPath = path.join(__dirname, '../data', file );
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 (i.e., public) to users of this module. This uses the syntax from the original Node CommonJS module system.

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`

9 In `data-provider.js` edit the following line.

```
module.exports = {
   filename: file,
   data: stocks
 };
```

Many modules provide access to multiple objects, and this example illustrates how you would allow this.

10 From `stocks-api.js`, edit the following line and test.

```
const provider = require('./scripts/data-provider.js');
const stocks = provider.data;
```

The variable `provider` contains the object assigned to `module.exports`.

Exercise 13a.11 — Route Handling in a Module

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

This example module will handle all the route handling for our API. It will export an object containing the different route handlers.

2 In this file add the following structure.

```
/* Module for handling specific requests/routes for stock data  */
const provider = require('./data-provider.js');
const stocks = provider.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 4 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 = (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 matches = stocks.filter(s => symbolToFind ===
                                          s.symbol);
      // return the matching stock
       resp.json(matches);
  });
};
```

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`, remove and add code so it appears as follows:

```
const path = require('path');
const express = require('express');

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

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

// set up route handling
const router = require('./scripts/stock-router.js');
router.handleAllStocks(app);
router.handleSingleSymbol(app);
router.handleNameSearch(app);

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

7 Test 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 now only about 20 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.12 — Adding in Some Error Handling

1 In `stock-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.

Currently, the file name for the JSON data file used in this API is hard-coded. A better approach would be to move the filename to an environment file; the file name can then be changed (perhaps to use an alternative one) without requiring the intervention of a programmer.

Exercise 13a.13 — Environment File Support

1 Create a file named `config.env` and add the following content to it.

```
PORT=8080
DATAFILE=stocks-complete.json
```

This file can be called anything, but needs to have the .env extension.

2 Modify `stocks-api.js` as follows.

```
let port = process.env.PORT;
app.listen(port, () => {
    console.log("Server running at port= " + port);
});
```

3 Modify `data-provider.js` as follows:

```
// for now, we will get our data by reading the provided json file
const file = process.env.DATAFILE;
```

4 Run node, but to use the environment file, you must run it using the appropriate flag:

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

Note: Prior to Node 20, you had to use an external package (dotenv) to get access to a file that had to be called .env.

Test Your Knowledge #1

1. Using the lab exercises as a guide, create a new node api named `artist-server.js`.

2. Your server will use the provided file `artists.json` in the `data` folder. You can examine the file `artist-single.json` to more easily see the structure of a single artist object.

3. Create the following routes. Be sure to return JSON error message rather than an empty array when there are no matches for the specified criteria.

`/api/artists`	Returns JSON for all artists
`/api/artists/id`	Return JSON for single artist with the supplied `ArtistId`.
`/api/artists/nationality/value`	Returns JSON for all artists with the specified `Nationality` value. This should be case insensitive.
`/api/artists/name/value`	Returns JSON for all artists whose `Lastname` starts with the specified `LastName` value. This should be case insensitive.

4. Test your work with the following requests:

```
localhost:8080/api/artists
localhost:8080/api/artists/101
localhost:8080/api/artists/3043403
localhost:8080/api/artists/nationality/france
localhost:8080/api/artists/nationality/sdfhds
localhost:8080/api/artists/name/ca
localhost:8080/api/artists/name/gdfgfdg
```