LAB 20a

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 respond to routes using Express
- How to create an API that implements CRUD functionality

Approximate Time

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

Fundamentals of Web Development, 2nd Ed

Randy Connolly and Ricardo Hoar

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

Date Last Revised: Feb 16, 2018

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. This lab has additional content contained within a folder named public. You will need to copy/upload this folder into your eventual working folder/workspace.

Exercise 20a.1 — INSTALLING 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 (for instance Cloud9), Node is likely already installed in your workspace.

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
   ```

   The second command will display the version number of npm, the Node Package Manager which is part of the Node install.

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. Run this file in node via the following command:

   ```
   node hello.js
   ```

   Not the most amazing program but it's a start!

Exercise 20a.2 — CREATING SIMPLE SERVER APPLICATION IN NODE

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 modules.
   A 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
*/
var http = require('http');

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

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

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

2 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.

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/ (the default port is 8080). If using a server-based Node environment, then you will have to request using the appropriate server URL. In an environment such as Cloud9, simply use the Preview the Running Application menu.

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.

4 Try modifying the URL path in the browser by adding content after the http://localhost/.

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 hello server.

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

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

It should display nothing (or some type of error message) since our server.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 20a.3 — Creating A Static File Server

1 Upload or copy the folder named public to your development location.

2 Create a new file named file-server.js.

3 Add the following code to this new file:

```
/* These additional modules allow us to process URL
   paths as well as read/write files */

var http = require("http");
var url = require("url");
var path = require("path");
var fs = require("fs");
```

4 Keep expanding this file by adding the following helper functions:

```
// outputs an HTTP 404 error
const output404Error = function(response) {
    response.writeHead(404, {"Content-Type": "text/html"});
    response.write("<h1>404 Error</h1>\n");
    response.write("The requested file isn't on this machine\n");
    response.end();
}
// outputs an HTTP 500 error (using arrow syntax)
const output500Error = (response, err) => {
    response.writeHead(500, {"Content-Type": "text/html"});
    response.write("<h1>500 Error</h1>\n");
    response.write(err + "\n");
    response.end();
}
```

Just to remind you of arrow syntax, the second function is defined using arrow syntax.

5 Now add the following code to create the server then save.

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

    // get the filename from the URL
    var requestedFile = url.parse(request.url).pathname;
    // now turn that into a file system file name by adding the current
    // local folder path in front of the filename
    var ourPath = process.cwd() + "/public";
    var filename = path.join(ourPath, requestedFile);
    console.log(filename);

    // check if it exists on the computer
    fs.exists(filename, function(exists) {
        // if it doesn't exist, then return a 404 response
        if (! exists) {
            output404Error(response);
            return;
        }

        // if no file was specified, then return default page
        if (fs.statSync(filename).isDirectory()) filename += '/index.html';

        // file was specified then read it and send contents to requestor
        fs.readFile(filename, "binary", function(err, file) {
            // maybe something went wrong ...
            if (err) {
                output500Error(response, err);
                return;
            }
            // based on the URL path, extract the file extension
            const ext = path.parse(filename).ext;

            // specify the mime type of file via header
            var header = {'Content-type' : mimeType[ext] || 'text/plain' };
            response.writeHead(200, header );

            // output the content of file
            response.write(file, "binary");
            response.end();
        });
    });

});

// 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 Run the following command:

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

7 In a browser request the page.

This should display the file index.html

8 Try requesting the other files in the public folder by adding the file name to the existing path in the browser (see Figure 20a.1).

9 When finished, be sure to stop the program using Ctrl-C in the terminal.

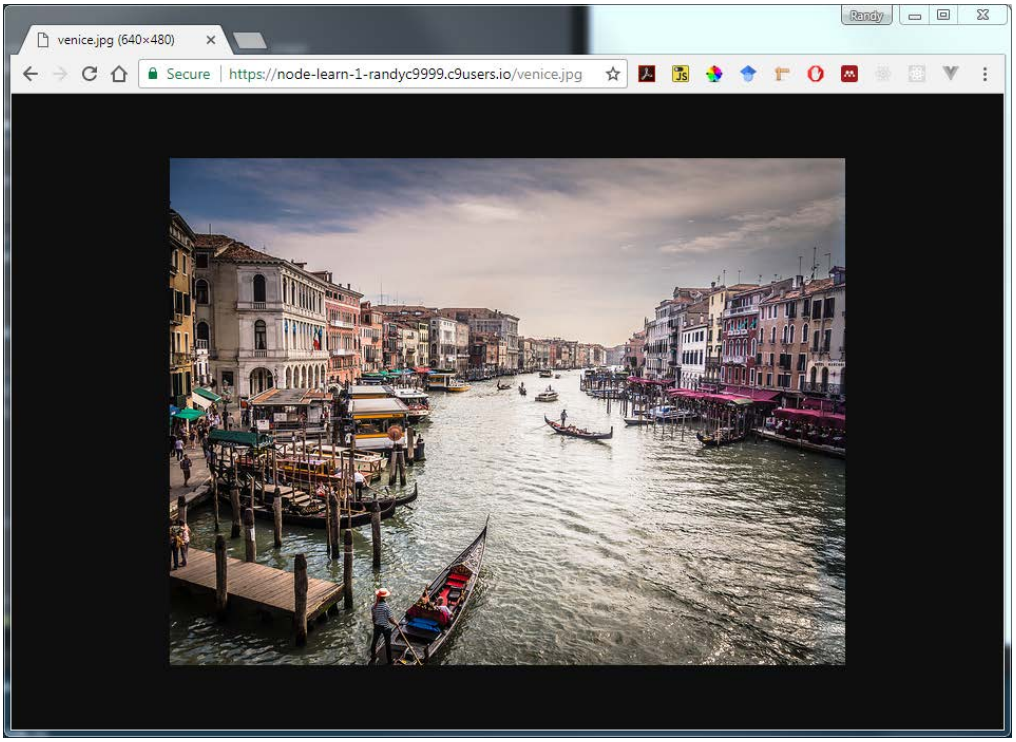

Figure 20a.1 – Running the file server

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 20a.4 — 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 information 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.js


your name

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 –save express
```

This downloads (from npmjs.com) the express package and, thanks to the –save flag, 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. What this command does, is tells npm to see if there are new versions of any of the dependent modules, and if there is, download and install them.

Exercise 20a.5 — Creating a JSON Web Service

1. Create a new file named stocks-simple.js.
2. Add the following code to this new file:

```
// first reference required modules
var fs = require('fs');
var path = require('path');
var parser = require('body-parser');
var 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
var jsonPath = path.join(__dirname, 'public',
                         'stocks-simple.json');
var jsonData = fs.readFileSync(jsonPath, 'utf8');
// convert string data into JSON object
var stocks = JSON.parse(fs.readFileSync(jsonPath, 'utf8'));

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

// tell node to use json and HTTP header features in body-parser
app.use(parser.json());
app.use(parser.urlencoded({extended: true}));
```

This code reads in the JSON file and sets up express

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

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

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

5. Test by running (via `node stocks-simple`) and viewing in browser.

   This should display the contents of the JSON file.

To make this web service more useful, you will need to add **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 20a.6 — Adding Additional Routing

1. Make a copy of stocks-simple.js and call it stocks.js.
2. Modify the filename as follows:

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

4. Run the file and test in browser.
5. Add the following additional route definition.

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

6 Test by making the following requests:

`https://your-domain-here/stock/AMZN`

`https://your-domain-here/stock/name/alph`

You may have to change the https to http depending on your environment. You should see the relevant JSON as shown in Figure 20a.2.

Figure 20a.2 – Testing the routes

Exercise 20a.7 — Adding File Serving

1 Add an additional route as follows:

```
// handle requests for static resources
app.get('/site/:filename', function (req, res) {
    var options = { root: path.join(__dirname, '../public/') };
    res.sendFile(req.params.filename, options, function (err) {
        if (err) {
          console.log(err);
          res.status(404).send('File Not Found')
        }
        else {
          console.log('Sent:', req.params.filename);
        }
    });
});
```

2 Test by making the following request:

`https://your-domain-here/site/tester.html`

This static file is included in the starting files. If you don't have it, add it (and tester.js) to the root folder of your project.

Exercise 20a.8 — Creating Your Own Modules

1. Create a new subfolder named routes.
2. In that folder, create a new file named file-router.js.
3. In this file add the following code:

```
var path = require('path');
module.exports = {


};
```

4. Cut the code from step 1 of the previous exercise and paste it into our new file inside of a new method shown below:

```
module.exports = {

    defineRouting: function(app) {

        app.get('/site/:filename', function (req, res) {
            var options = { root: path.join(__dirname, '../public/') };
            res.sendFile(req.params.filename, options, function (err) {
                if (err) {
                  console.log(err);
                  res.status(404).send('File Not Found')
                }
                else {
                  console.log('Sent:', req.params.filename);
                }
            });
        });
    }
};
```

5. In stocks.js, add the following code near the top of the file:

```
// reference our own modules
var staticFileRouter = require('./routes/file-router.js');
```

6. In stocks.js, replace the static resource route with a call to the function in our new module:

```
staticFileRouter.defineRouting(app);
```

7. Save both files and test.

   The script should work the same but be more modular now.

8. In the routes folder, create a new file named name-router.js.
9. In this file, move the substring route code from Exercise 20a.6 so your code looks similar to the following:

```
module.exports = {
    defineRouting: function(stocks, app) {
        // return all the stocks whose name contains the supplied text
        app.route('/stock/name/:substring')
```

```
            .get(function (req,resp) {
                // change user supplied substring to lower case
                var substring = req.params.substring.toLowerCase();
                // search the array of objects for a match (arrow syntax)
                var matches = stocks.filter( (obj) =>
                        obj.name.toLowerCase().includes(substring) );
                // return the matching stocks
                resp.json(matches);
            }
        );

    }
};
```

10 In the routes folder, create a new file named symbol-router.js.

11 In this file, move the symbol route code from Exercise 20a.6 so your code looks similar to the following:

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

10 In stocks.js, add the following code near the top of the file:

```
// reference our own modules
var staticFileRouter = require('./routes/file-router.js');
var nameRouter = require('./routes/name-router.js');
var symbolRouter = require('./routes/symbol-router.js');
```

11 In stocks.js, replace the just-cut substring route with a call to the function in our new module:

```
nameRouter.defineRouting(app);
symbolRouter.defineRouting(stocks, app);
```

12 Save both files and test.

The script should work the same but be more modular now.

IMPLEMENTING CRUD BEHAVIORS

For JavaScript intensive applications, it is common for web services to provide not only the ability to retrieve data, but also create, update, and delete data as well. Since REST web services are limited to 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 the HTTP verb with the 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 exercise you will add this CRUD functionality. A form with JQuery code 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 MongoDB.

Exercise 20a.9 — 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. Notice that it uses the jQuery $.ajax() function to make PUT, POST, and DELETE requests. Notice also that the code passes JSON data from the client to the server.

2 In the symbol-router.js file, add the following code (some code omitted):

```
// the lodash module has many powerful and helpful arrary functions
var _ = require('lodash');

module.exports = {
  defineRouting: function(stocks, app) {
    app.route('/stock/:symbol')
        .get(function (req,resp) {
            ...
        })
        // if it is a PUT request then update specified stock
        .put(function (req,resp) {
            console.log('put request');
            var symbolToUpd = req.body.symbol.toUpperCase();

            // use lodash module to find index for stock with this symbol
            let indx = _.findIndex(stocks, ['symbol', symbolToUpd]);
            // if didn't find it, then return message
            if (indx < 0) {
                console.log('Symbol not found='+symbolToUpd);
                resp.json({ message: 'Symbol 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({message:'Stock ' + symbolToUpd + ' updated!' });
                console.log('Symbol updated ='+symbolToUpd);
            }
        }
    );
  }
}
```

3 Save and test by requesting the 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.

4 In a separate tab, make a GET request for the same symbol:

`https://your-domain-here/stock/AMZN`

It should contain the updated values.

5 Our code just 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 a future lab, we will make changes persistent by recording them in a MongoDB database.

Exercise 20a.10 — Adding Insert and Delete Support

1 In the symbol-router.js file, add the following code (some code omitted):

```
...
module.exports = {
  defineRouting: function(stocks, app) {
    app.route('/stock/:symbol')
        .get(function (req,resp) {
            ...
        })
        .put(function (req,resp) {
            ...
        })
        // if it is a post request then insert new stock
        .post(function (req,resp) {
            console.log('post request');

            stocks.push({
                symbol: req.body.symbol,
                name: req.body.name,
                SEC: req.body.sec,
                sector: req.body.sector,
                subIndustry: req.body.subIndustry,
                address: req.body.address,
                dateAdded: req.body.dateAdded,
                CIK: req.body.cik,
                frequency: req.body.freq
            });
            resp.json({ message: 'New stock ' + req.body.symbol +
                           ' added!' });
        } )

        // if it is a delete request then delete specified stock
        .delete(function (req,resp) {
            var symbolToDel = req.params.symbol.toUpperCase();
            console.log('delete request symbol=' + symbolToDel);
            // use lodash function to remove matching element from array
            _.remove(stocks, {symbol: symbolToDel});
            resp.json({ message: 'Stock ' + symbolToDel + ' deleted!' });
        }
      );
   }
}
```

2 Save and test by requesting the tester.html file.