ELECTRICAL AND INFORMATION ENGINEERING

University of the Witwatersrand, Johannesburg

Software Development III

Laboratory 2 — JavaScript and Node.js

In this lab, we will go over the fundamentals of the JavaScript language. We assume that you are familiar with C++ and so we highlight some of the key differences with respect to C++. A brief introduction to Node.js as a web server is also provided.

1 Install Node.js

In this course we will be using the LTS (Long Term Support) version of Node.js. Download and install Node for your operating system.

Node is a JavaScript run-time environment which allows us to execute JavaScript code *outside of the browser*. This is useful because although you presumably already have a JavaScript compiler installed on your computer (via Chrome, Firefox, or another browser), it is handy to be able to run JavaScript from within an IDE and make use of the IDE's auto-completion, debugging, and other supporting features.

Note, Node.js is typically run on a server for executing server-side scripts which generate web pages. This will be discussed in more detail in Section 14.

Check that node is installed correctly by typing the following in the command line:

```
node -v
```

2 Run Your First JavaScript Program

Make sure that you already have installed Visual Studio Code, as described in the first laboratory.

Exercise 1

Before starting, note that you should create a separate JavaScript file for each of the exercises given in this lab.

Now let's create our first JavaScript program. Type the following in a new file and save it as `hello.js`.

```
const message = 'Hello World!'
console.log(message)
```

Now let's run the code. Right-click on the file in VS Code's Explorer pane (Ctrl-Shift-E) and choose `Open in Terminal`. At the prompt, type:

```
node hello.js
```

and you should see "Hello World!" being printed. Notice that (unlike C++) we didn't need any `main` function to run our code. JavaScript, like most scripting languages, runs files line-by-line from the top of the file.

3 JavaScript Syntax and Style

```
let sum = 0;

for (let i = 0; i < 9; i++) {
  sum += 1;
}

console.log(sum);

if (sum % 2 == 0) console.log("Sum is even");
else console.log("Sum is odd");
```

Listing 1: JavaScript has a similar syntax to C++

Viewing Listing 1, we see that JavaScript's syntax is very similar to C++'s. `for`-loops and `if`-statements are written in an identical fashion. Curly braces define variable scope as we are used to.

JavaScript is "relaxed" when interpreting syntax and so although the code that is given has semi-colons it is entirely possible to exclude them as they are optional. Single quotes can also be used for declaring string literals. In JavaScript `'=='` is the loose equality operator. This can be difficult to use correctly, so the strict equality (`===`) and inequality (`!==`) operators are generally preferred. Most of these stylistic decisions are minor so it is best to just choose one approach and stick to it. In this course we will follow the conventions of StandardJS, some of which are evident in Listing 2.

```
let sum = 0                     // no semi-colons

for (let i = 0; i < 9; i++) {
  sum += 1
}

console.log(sum)
                                // use strict equality comparisons: '==='
if (sum % 2 === 0) console.log('Sum is even')
else console.log('Sum is odd')  // single quotes for strings
```

Listing 2: JavaScript formatted according to StandardJS

3.1 Linting

Manually formatting code to conform to a specific style is laborious so it is best to automate this task. Programs that check code for stylistic issues and common syntactical errors are known as *linters*. We will be using the StandardJS linter to format our JavaScript files.

First install the StandardJS package (what we would call a library in C++) using the Node Package Manager (NPM). NPM is automatically installed as part of the Node installation (Section 1). Type the following in the terminal (if your terminal does not have focus, then you can change to it using the shortcut Ctrl-`, which is very handy).

```
npm install standard --global
```

Now install the StandardJS extension for VS Code. It is helpful to set the extension to automatically format your code when it is saved. To do this, go to `File|Preferences|Settings` and type "standard" in the search box. Look for the setting `Auto Fix On Save` and check the tick box.

Exercise 2

Now test this out by typing out Listing 1 and then saving the file. It should be formatted to look like Listing 2. StandardJS is also helpful in that it will identify unused variables and other anomalies. These will be listed in VS Code's `PROBLEMS` pane (Ctrl-Shift-M).

4 Variables

In JavaScript, variables are declared mostly the same way as in C++. However, because JavaScript is loosely typed, the variable type is not given when the variable is declared *and the variable need not be initialised*. In C++ when you use **`auto`**, you are required to initialise the variable so that its type can be deduced.

4.1 Variable Declaration

The **`let`** and **`const`** keywords are used to declare variables. Use **`let`** for ordinary variables and **`const`** for constants. Remember to minimize variability by using **`const`** whenever possible. Variable names can only start with letters, underscores and dollar signs. Below are valid declarations of variables:

```
let a
const b = 3
let $c, _d
let e = 4, f = 'five'
```

In this example, the values of `a`, `$c` and `_d` are **`undefined`** while `b`, `e` and `f` have been initialised with values. `b` is a constant and cannot be changed.

4.2 Strict Mode

JavaScript was originally intended for writing very small web scripts and so ease-of-use took precedence over code quality. An example of this is that if you forget to declare a variable and simply start using it, the compiler will happily create a global variable for you and not complain. This is known as a *leaked global* and is undesirable for obvious reasons in larger applications. To prevent this from happening it is necessary to run JavaScript in strict mode. To enable this, the first line of your JavaScript files should always contain the special directive: `'use strict'`

Strict mode also prevents us from naming variables after keywords that are reserved for future use, such as **`public`**.

4.3 The `var` Keyword

In older JavaScript code, and in the MDN JavaScript documentation which will be referred to from time to time, you will see variables being declared with the **`var`** keyword. Writing code using **`var`** declarations is far more error-prone than using **`let`** and **`const`**. This is because variables declared using **`var`**:

- can be redeclared without warning,
- are function scoped, not block scoped, as we are used to, and
- can be used prior to their declaration because of variable hoisting.

Therefore, we will only be declaring variables with **`let`** and **`const`**.

5 Data Types

In JavaScript, there are seven data types. Six data types are primitive:

- **`Boolean`**: Has two values: true and false, just like C++.
- **`Number`**: This type includes both integers and decimal values.
- **`String`**: Predictably, these are sequences of characters.
- Special types: **`Null`** and **`Undefined`**. These types can only be set to a single value: **`null`** and **`undefined`** respectively. The compiler will assign **`undefined`** to variables that are not initialised. We should assign **`null`** to a variable if we intentionally wish to clear it.
- **`Symbol`**: A symbol is created by invoking a function which produces an anonymous, unique value. This is used in more advanced JavaScript programming and we won't discuss it further in this lab.

Lastly, there is the **`Object`** type. In JavaScript an object is a list of key-value pairs (like a map in C++). An array is also an object which may contain elements which are *not necessarily of the same type* (unlike C++). Surprisingly, *functions are regular objects* as well. This should be the first hint that JavaScript, although syntactically similar to C++, is fundamentally different. Functions enjoy an equal, if not privileged status, when compared to objects. Objects, arrays and functions will be discussed in much more detail below.

5.1 Aside: Template Strings

At this point it is worth discussing template strings which allow variables and/or code to be embedded within a string literal. This makes the resulting string much more readable.

```
'use strict'

const a = 5
const b = 10

// an ordinary string
console.log('Fifteen is ' + (a + b) + ' and not ' + (2 * a + b) + '.')

// using a template string - note the back-ticks
console.log(`Fifteen is ${a + b} and not ${2 * a + b}.`)
```

Listing 3: Using template strings

In Listing 3 we can see that templates strings are declared with the back-tick symbol (`) and they allow expressions (code) to be embedded within them using `${expression}`.

5.2 Weak Typing

As mentioned before, JavaScript is loosely typed. Types are only decided when a value is assigned to a variable and they can be changed on-the-fly based on the operations done to the variable.

Exercise 3

Try typing the following into a `.js` file and running it in the terminal:

```
'use strict'  // don't forget this, it won't be shown in future code samples

let a = 'hello'
console.log(`a = ${a}, "a" is ${typeof a}`)

a = 5
a += 2
console.log(`a = ${a}, "a" is ${typeof a}`)

a = true
console.log(`a = ${a}, "a" is ${typeof a}`)
```

The **`typeof`** operator is explained in the MDN reference. Notice how the type of `a` changes.

6 Functions

In JavaScript, *functions are treated as values* which means that we can assign them to variables. This is illustrated in line 1 of Listing 4. On the left-hand side of the equals sign is the variable `add` which is being assigned to the *function expression* on the right-hand side. The function is anonymous in that it has not been given a name and can therefore only be accessed through the variable assigned to it.

```
1  const add = function (a, b) {   // function expression assigned to 'add'
2    return a + b
3  }
4
5  const result = add(2, 3)        // execute the function
6  console.log(result)
7
8  const anotherAdd = add
9  console.log(anotherAdd(12, 4))  // execute it again
```

Listing 4: An anonymous function expression

The function is first run on line 5 where the opening and closing braces are encountered. On line 7 another variable (`anotherAdd`) is assigned the function expression and the function is executed once more on line 9.

No types are given for the function parameters or the return type. If the function is called and a parameter is missing then the parameter is **`undefined`**. For example, if the function was called using `add(2)` then `b` would be **`undefined`**. Also, if a function does not explicitly return a value then the return value is **`undefined`**. It is also worth noting that function parameters can have default values.

Exercise 4

Create and test a function which converts temperatures from Fahrenheit to Celsius.

6.1 Function Declarations

JavaScript also allows for *function declarations* in which functions are named and declared in similar fashion to C++. To find out more about this refer to the documentation on function declarations. We will typically use function expressions.

7 Objects

JavaScript objects stored keyed collections of data. Listing 5 shows how to create a `student` object and access its properties.

```
let student = {
  name: 'Kwezi',
  studentNumber: 453528
}

console.log(student) // print the entire object

// access the object's properties
console.log(`${student.name}'s student number is ${student.studentNumber}`)
```

Listing 5: Creating a `student` object and accessing its properties

Functions can be written to manipulate objects, as shown in Listing 6.

```
const addAge = function (theStudent, age) {
  theStudent.age = age // add a new property called age to the student
}

addAge(student, 20)
```

Listing 6: Adding a new property to `student`

Verify that the code in Listing 6 works as expected.

Exercise 5

Write some JavaScript to create a `course` object which has a single `courseCode` property. Set the `courseCode` to the string `'ELEN4010'`.

Now create function which takes in a course object as an parameter and adds the property `yearOffered`. The `yearOffered` property should be determined from the course code.

Fourth year courses all contain the numerals “40”, third year courses contain “30”, and so on. Hint: use the string class’s `includes` method.

Create another function which takes in a course as a parameter and returns a string summarizing the course information. For ELEN4010 it should return: “ELEN4010 is offered in year 4.”

Test your solution by creating course objects from other years of study.

7.1 Pass by Value and Pass by Reference

As in C++, primitive types (see Section 5) are passed by value, that is, a function receives a copy of the variable which is passed to it.

Objects and arrays (Section 8), on the other hand, are passed by reference. It is helpful to *visualize JavaScript references as C++ pointers* — not C++ references.

In Listing 6 the function accepts a student object as a parameter. Within the function `theStudent` is a *reference* to the student object containing the name “Kwezi”. This means that any changes made to the properties of the student object within the function will be reflected outside of the function. This was shown when adding the `age` property to `student`.

It is important to note that *the reference itself is passed by value*. In other words, reassigning a reference within a function will not affect the object passed in. This is illustrated in Listing 7.

```
const setStudentToEmptyObject = function (theStudent) {
  theStudent = {} // assign the reference to an empty object
}

setStudentToEmptyObject(student) // has no effect on 'student'
```

Listing 7: Object references are passed by value

7.2 Object Methods

In C++ we talk about an object having member functions. In JavaScript these functions are known as *methods*. Methods are very easy to create by simply assigning an object’s key to a value which is a function. Listing 8 demonstrates this. Notice how the `getSummary` key has been assigned a function returning a summary of the student. The **`this`** keyword refers to the object on which the function is being invoked and allows us to access the other properties of the object. In C++ **`this`** can be used in member functions but it is not required.

```
let student = {
  name: 'Kwezi',
  studentNumber: 453528,
  // 'this' refers to the object on which the method is invoked
  getSummary: function () {
    return `${this.name}'s student number is ${this.studentNumber}`
  }
}
```

Listing 8: Adding a method (function) to `student`

Exercise 6

Create an `account` object and provide methods for depositing and withdrawing money from the account. Also provide a method which returns a summary of the account's transactions. This method should return a string similar to the following: "This account has a balance of R 200. There have been deposits totalling R 300 and withdrawals totalling R 100." You need to decide on what properties the `account` object should have.

8 Arrays

Arrays are an important object type in JavaScript. As in C++, they are zero-indexed. Unlike C++, they are heterogeneous which means that they can contain different types. When declaring as array, the elements are separated by commas as shown below:

```
let mixed = ['hello', 5.5, 7, false]
```

Arrays have the performance characteristics of a C++ `vector` because they are stored in contiguous memory (if used correctly). Therefore, insertions and deletions at the end of the array are fast.

Exercise 7

The MDN reference on Arrays provides a comprehensive guide to the various Array methods. Use the MDN reference to predict the output of the Listing 9.

```
let numbers = [76, 55.7, 89, 37.5, 61]
numbers.push(19)
numbers.unshift(61)
numbers[1] = 12
numbers.splice(3, 1, 99)

console.log(numbers)
console.log(numbers[7])
```

Listing 9: Exercising some of Array's methods

9 Higher-Order Functions and Callbacks

Higher-order functions are functions that operate on other functions, either by taking them in as arguments or by returning them. This is fairly common in JavaScript because functions are values just like numbers and strings. Suppose we wish to log to the console every element in our `numbers` array, we could do it in the usual fashion as shown in Listing 10.

```
for (let index = 0; index !== numbers.length; index++) {
  console.log(numbers[index])
}
```

Listing 10: Using a typical `for` `loop` to display an array's contents

However, a more idiomatic way of expressing this in JavaScript is to use a *callback function* which is called by Array's `forEach` method. This is shown in Listing 11. In this approach, the `forEach` method calls the function that is supplied to it once for each element in the array, and it passes the element in question as an argument. Notice how the function is defined at the point at which the `forEach` method is called.

A callback function, or more simply a *callback*, is any function that is passed as an argument to other code that is expected to call it (execute it) at some point in time. It is not called directly as is done with normal functions.

```
numbers.forEach(function (element) {
  console.log(element)
})
```

Listing 11: Using `forEach` and a callback function to display an array's contents

Exercise 8

Write a function that applies an arbitrary function to each element of an array and places the result in a new array. Your function signature should be:

```
const map = function (functionToApply, array)
```

Now test your `map` function with a function that squares the contents of an array which contains numbers.

In fact, the `map` method already exists for JavaScript Arrays so there is no need to write your own. Check that when you use Array's `map` method the results match your own `map` function.

10 Object Equality

In C++ primitive types can be directly compared but we are required to provide an equality operator (`operator==`) in order to compare objects. JavaScript also allows for primitive types to be directly compared. In Listing 12 we can use `indexOf` to search an array of primitive types. `indexOf` returns `-1` if the search element is not found.

```
const numbers = [76, 55.7, 89, 37.5, 61]

console.log(numbers.indexOf(89)) // prints 2
console.log(numbers.indexOf(234)) // element not found: prints -1
```

Listing 12: Searching for a primitive type in an array

When searching for elements using `indexOf` the element being searched for is compared to each of the array elements using strict equality (`===`). No element is found in Listing 13. This is because when two *objects are compared the equality operator compares their references* (think pointers in C++) not the actual objects' properties.

```
const electives = [
  {
    courseCode: 'ELEN4010',
    yearOffered: 4
  },
  {
    courseCode: 'ELEN4001',
    yearOffered: 4
  },
  {
    courseCode: 'ELEN4020',
    yearOffered: 4
  }]

console.log(electives.indexOf({ // no element found
  courseCode: 'ELEN4001',
  yearOffered: 4
}))
```

Listing 13: `indexOf` cannot be used to search for objects

To see this demonstrated more clearly, refer to Listing 14

```
// the empty objects' properties are not compared,
// their references are - '===' returns false
console.log({} === {})

// equal object references - '===' returns true
const someObject = {}
const otherObject = someObject
console.log(someObject === otherObject)
```

Listing 14: Comparing objects using strict equality

So in order to find an object in an array, we need to make use of Array's `findIndex` method. The `findIndex` method accepts a callback function which is called for each element in the

array. The callback function needs to be a *predicate* which means that it must return a **boolean** value: **true** if the search element matches the array element, and **false** otherwise. It returns the index of the first matching element that is found. This is illustrated in Listing 15.

```
console.log(electives.findIndex(function (element) { // prints 1
  return element.courseCode === 'ELEN4001' &&
         element.yearOffered === 4
}))
```

Listing 15: Using `findIndex` to find an object within an array

Exercise 9

Create a function to delete a student from the array of students given in Listing 16.

```
let students = [
  {
    name: 'Kwezi',
    studentNumber: 453528,
    yearOfStudy: 4
  },
  {
    name: 'Pieter',
    studentNumber: 454345,
    yearOfStudy: 3
  },
  {
    name: 'Jade',
    studentNumber: 678343,
    yearOfStudy: 4
  },
  {
    name: 'Kiren',
    studentNumber: 567893,
    yearOfStudy: 4
  }
]
```

Listing 16: Students array

Your function should take in the student to be deleted and the array of students. It should return the modified array. If the student cannot be found then the array should be returned unmodified. A call to the function is shown below.

```
const modifiedArray = deleteStudent({
  name: 'Kiren',
  studentNumber: 567893,
  yearOfStudy: 4
}, students)
```

Your solution must not contain any `for` or `while` `loops`. Remember that arrays are passed by reference so modifications to their contents within a function will be reflected outside of that function.

11 Arrow Functions

Arrow functions offer a very compact way of defining functions. This can considerably improve readability when functions are expressed in line as arguments for higher-order functions.

```
                  // function (parameters) { function body}
const squareNormal = function (num) { return num * num }

                   // (parameters) => { function body }
const squareShorter = (num) => { return num * num }

                //     parameter   => single-line function body
                // OR (parameters) => single-line function body
const squareShortest = num => num * num

// all functions are called in the same way
console.log(squareNormal(2))
console.log(squareShorter(3))
console.log(squareShortest(4))
```

Listing 17: Shorter and shorter functions

The most compact syntax (`squareShortest`) can only be used for functions containing a single `return` statement as the function body (the `return` keyword is omitted). Arrow functions do not bind to `this` and therefore are not suited for use as object methods.

Watch this video by mpj which gives an excellent overview of arrow functions.

Exercise 10

Given the code in Listing 18, use arrow functions and Array's `filter` method to produce an array containing only fourth year students doing ELEN4010 as an elective. The resulting array should only contain Kwezi and Kiren. Your solution must not contain any `for` or `while` `loops`.

```
const electiveOne = {
  courseCode: 'ELEN4010',
  yearOffered: 4
}

const electiveTwo = {
  courseCode: 'ELEN4001',
  yearOffered: 4
}

const electiveThree = {
  courseCode: 'ELEN4020',
  yearOffered: 4
}

const electiveFour = {
  courseCode: 'ELEN4017',
  yearOffered: 4
}

let students = [
  {
    name: 'Kwezi',
    studentNumber: 453528,
    yearOfStudy: 4,
    electives: [electiveOne, electiveTwo, electiveThree]
  },
  {
    name: 'Pieter',
    studentNumber: 454345,
    yearOfStudy: 3,
    electives: [electiveOne, electiveTwo, electiveFour]
  },
  {
    name: 'Jade',
    studentNumber: 678345,
    yearOfStudy: 4,
    electives: [electiveTwo, electiveThree, electiveFour]
  },
  {
    name: 'Kiren',
    studentNumber: 567893,
    yearOfStudy: 4,
    electives: [electiveOne, electiveTwo, electiveThree]
  }
]
```

Listing 18: Students and their electives

12 Asynchronous Functions

In C++, we are used to functions that return a value once they have completed their task. A result of these types of functions is that we have to wait for a function to finish processing before we can continue to the next function. This is called **blocking** code, because it blocks all other operation while it operates.

Imagine that you needed to read a file in for a user. While your file is being read in, you wouldn't even be able to display a progress bar because your code is busy waiting for the file read operation to finish. This makes your program look unresponsive. (A similar problem occurs on websites where images are loaded. If the load operation is blocking, the website would be completely unresponsive while loading each image one-by-one).

Let's see a simple demonstration:

```
const sleep = function (sleepDuration) { // A function that blocks
  const now = new Date().getTime()
  while (new Date().getTime() < now + sleepDuration) { /* do nothing */ }
}

console.log('Starting')

sleep(10000)

console.log('Done, doing other things')

for (let i = 0; i < 10; i++) {
  console.log(i)
}

console.log('Done with other things')
```

Listing 19: Blocking `sleep` function

Notice how long you had to wait.

In order to fix this issue, JavaScript uses **non-blocking** function calls. It achieves this by using callbacks. Instead of returning a value, a function returns immediately and takes an extra callback function as an parameter. This callback function is called when the function completes with the results as an parameter and contains code to be run after the function completes. By returning immediately, the function let's the main code continue running while it completes its task in the background. Let's try this out:

```
console.log('Starting')

// setTimeout(callback, duration) is an asynchronous version of sleep
setTimeout(() => {
  console.log('Done Waiting')
}, 10000)

console.log('Done, doing other things')

for (let i = 0; i < 10; i++) {
  console.log(i)
}

console.log('Done with other things')
```

Listing 20: Non-blocking `sleep` function

Notice that your code did the rest of its work before the `setTimeout` function finished.

13 NPM and Modules

It would be preferable to be able to structure our program using different files. For this purpose, `nodejs` provides the module system.

Let's learn about it using a demonstration: Create files called `main.js` and `mod.js` In `main.js`, put:

```
console.log("Loading a module");
require("./mod");
console.log("Done");
```

In `mod.js`, put:

```
console.log("I'm inside a module!");
```

Now run `main.js` using node.

So, modules are scripts that we can call from our main file. However, we can use them to implement libraries (similarly to C++'s include) using the exports functionality. Let's learn using a demonstration again: Edit `main.js` to say:

```
console.log("Loading a module");
let mod = require("./mod");
console.log("Mod:", mod);
console.log("Done");
```

Edit `mod.js` to say:

```
console.log("I'm inside a module!");
module.exports = {
    some: "module",
    number: 2
};
```

Now run `main.js` using node.

Notice that `require` returns the value of `module.exports` from `mod.js` once its done running. Remember, unlike a function return, `require` will run the whole script, even if module.exports was assigned in the middle. It will only return when the script finishes.

13.1 Built-in modules

`nodejs` ships with a few modules built into it. Let's try using the `File System` module to read a text file: Edit `main.js` to say:

```
let fs = require("fs");
fs.readFile("./ELEN4010.txt", "utf8", (err, data) => {
    // utf8 is the encoding of the file
    if (err) throw err; //callbacks usually get errors in this format
                        //err will either contain the error or null
    console.log(data); //if there weren't errors, send the data to the console
})
```

Create `ELEN4010.txt` and fill it with a short essay recounting your experiences in ELEN4010 so far (or anything else really, it doesn't matter) Now run `main.js` using node.

13.2 NPM Modules

Apart from the core module built into node. There is a repository of 3rd party modules called npm. It can be found at: www.npmjs.com. Modules from npm can be installed by calling `npm install <packagename>` from the terminal in your project directory.

CHALLENGE: Since we are too lazy to make essays for all the other courses, create a script using www.npmjs.com/package/dolor and the `fs` module to fill text files for your other courses with placeholder text.

14 Express

In this course we will be using an NPM module called `express`. It is a framework for using `node` as a web server.

To begin, create a new folder and `cd` into it using the terminal. Now run:

```
npm init
```

Follow the instructions to create a new NPM project in the folder (it also lets you manage your project using git and manage its npm dependencies). Make your entry file `index.js` and leave any fields you don't know blank.

Make sure to add a *.gitignore* file so you don't commit files you shouldn't. You can find one for NodeJS here: https://gitignore.io/api/node.

Now install `express` and save it to the package.json that npm created using:

```
npm install --save-exact express
```

If you are behind the Wits proxy, you have number of options to download packages using NPM. The easiest is to configure NPM with your proxy user name and password. Type the following:

```
npm set proxy http://students%5C<student-no>:<password>@proxyss.wits.ac.za:80
```

All special symbols have to be URL encoded so the "\" after the domain is encoded as "%5C". If there are any special symbols in your password remember to supply the URL encoded versions instead. See (https://www.w3schools.com/tags/ref_urlencode.asp) for a URL encoding reference.

The second approach is to use a program called `CNTLM` which will act as an intermediate proxy for the Wits network: http://cntlm.sourceforge.net/. You would then run the above command as follows:

```
npm --proxy http://localhost:3128 install --save-exact express
```

The third approach is possible if you are running on a Linux or Unix machine. Use the `proxychains` program which allows any command in the terminal to be run with proxy settings. Download it from https://github.com/haad/proxychains or your package manager and run it using the following command (once you have configured the program):

```
proxychains npm install --save-exact express
```

Now, create `index.js` and put the following into it:

```
let path = require('path');
let express = require('express');
let app = express();

app.get('/', function (req, res) {
  res.send('Hello World');
});

app.listen(3000);
console.log("Express server running on port 3000");
```

Run it using `node` and browse to 127.0.0.1:3000 in your web browser to see it working. Try browsing to 127.0.0.1:3000/about. Doesn't work? That's because we don't have a route for about.

14.1 Routing

Routes allow us to define how express responds depending on the path that they access. A routing function takes the form:

```
app.<request type>(<path>, callback);
```

We have one route defined in index.js on line 4. We can see that the route is for a GET request to the path "/".

Lets create a route for the about page and serve some html this time: First, create a folder called `views` in your project directory. Inside this directory, create `about.html` with the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>About</title>
    <meta charset="utf-8" />
</head>
<body>
    <h1>This website was made by:</h1>
    <p>
    (your name here)!
    </p>
</body>
</html>
```

Now let's define a route to the about page: In `index.js` add the following new route:

```
app.get('/about', function(req, res){
    res.sendFile(path.join(__dirname, 'views', 'about.html'));
});
```

You should also require `path`, it's a node core module.

Try running the your code and browsing to `127.0.0.1:3000/about` again.

14.2 Routing as a module

Putting all our routes into the main application script can be really inconvenient and messy. As an alternative, we can define our routes as a module using `express.router`. It is a class in express that implements routing and can be added to your app as middleware.

Before we make a routing module, let's move our routes to a router in `index.js`. First create a router in `index.js`, replace your routes with:

```
let path = require('path');
let express = require('express');
let app = express();
let mainRouter = express.Router();

mainRouter.get('/', function (req, res) {
  res.send('Hello World');
});

mainRouter.get('/about', function(req, res){
    res.sendFile(path.join(__dirname, 'views', 'about.html'));
});

app.use(mainRouter);

app.listen(3000);
console.log("Express server running on port 3000");
```

Verify that your website still works.

Now, create a file called `mainRoutes.js` with the following:

```
let path = require('path');
let express = require('express');
let mainRouter = express.Router();
mainRouter.get('/', function (req, res) {
  res.send('Hello World');
});

mainRouter.get('/about', function(req, res){
    res.sendFile(path.join(__dirname, 'views', 'about.html'));
});
module.exports = mainRouter;
```

In `index.js` remove the routes and change `mainRouter` to be required from `mainRoutes.js`. Verify it works using your browser.

15 Deploying Node.js to Azure

Now that you've set up your Node application locally, you will have to make changes so you can deploy and run it on the Azure *web-app* you created in lab 1. Azure will automatically detect that you are wanting to run a Node application but it still needs to know what file is the entry point to your program.

To do that you must edit the **`package`**`.json` file that was created when you ran '`npm init`'. Add a line to the `scripts` section declaring the `start` option as shown below. The sample text below is for a main JavaScript file called *index.js*. If you have a different file-name then replace `index` with the file-name **excluding the file extension (`.js`)**.

```
...
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index"
},
...
```

Note: Make sure you are working in the repository you used for your Azure deployment in lab 1. You may use another repository but then you will need to go through the Azure *web-app* part from lab 1 again.

To make it easier for us to know if the node server was deployed and launched successfully, alter your `mainRoutes.js` file to contain the code below for the main route.

```
mainRouter.get('/', function (req, res) {
  res.send('Hello World, I\'m Node.js');
});
```

When your Node application runs locally, it accepts requests on port *3000*. The standard port for the web (http and https) is 80. To make sure your application knows what port to use, replace `app.listen(3000);` with the line shown below.

```
let port = process.env.PORT || 3000;
app.listen(port);
console.log("Express server running on port", port);
```

On the Azure server there's an environment variable called `PORT` which is assigned the value *80*. '`process.env.PORT`' attempts to access this value and return it; however, if no such variable exists then the value *3000* is passed to `app.listen`. This way if you run your application locally it will still use port 3000 but once it is deployed onto your Azure instance it will automatically use port 80.

Before you push your changes to your remote (GitHub), navigate to your Azure web service URL and notice that it still shows the `html` file you created in lab 1. Also navigate to your local server for your new Node application. For lab 1 we set the *stack runtime* to *PHP 7*. For your NodeJS application we need to change it. Navigate to your web-app on the Azure Portal and go to the *Configurations* page under the *Settings* tab on the left.

Go to the *General Settings* tab on this page change the runtime stack to *Node* and the major and minor versions to *12 LTS* and then click save at the top of the page. Push your changes to GitHub and then continuously refresh your Azure web-page until you notice the new *Hello World* message you created above.

Your Node app is now running on Azure after automatically being deployed!