SENG365 Lab 1
Introduction to JavaScript, Node.js and APIs

SENG365

Ben Adams Morgan English

17th February 2023

Purpose of this Lab

This lab is split into two parts. In the first part we introduce JavaScript and Node. JavaScript is the default scripting language used in all standard web browsers (e.g., Chrome, Safari, Firefox, and Edge) and is therefore the language for implementing the majority of web applications. Node brings the power of JavaScript to the server-side of an application. Consequently, JavaScript can be used on both the client-side and the server-side of a web application. In the second part of this lab, we begin to look at how to use Node and Express to implement APIs.

This lab looks long, but it's mostly due to the amount of theory you need to be aware of to get started. To properly understand the concepts covered it is highly recommended you read the online resources discussed. Subsequent labs will be more practical and require less reading.

1 Overview of JavaScript

Mozilla provides a great introduction to what JavaScript is here: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction`. Notably the **What is JavaScript?** and **JavaScript and Java** sections.

JavaScript is a widely used cross-platform, object-oriented programming language that underpins modern web development on both the client and server side. Whilst you may not have directly interacted with or written JavaScript code yourself, almost every web application makes heavy use of JavaScript to provide functionality. For those interested you can visit a website in Chrome and open the dev tools (F12), navigating to the 'Sources' tab should show you many JS files though generally these have been minified[1] so it can be difficult to understand what's going on.

Throughout this course we will be using TypeScript, a superset of JavaScript (this will be discussed more in the next lab), for the majority of the exercises. Specifically we will be using the ES6[2] standard. If you are new to JavaScript and/or the ES6 standard then you should take some time now to familiarise yourself. There is an abundance of information available online:

- `https://developer.mozilla.org/en-US/docs/Web/JavaScript`
- `https://web.dev/learn/pwa/getting-started/`
- `https://www.w3schools.com/js/`
- `https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/`

As well as many books:

- `https://exploringjs.com/`

[1] See `https://www.cloudflare.com/learning/performance/why-minify-javascript-code/`
[2] See `http://es6-features.org/`

- https://leanpub.com/understandinges6
- https://leanpub.com/es6-in-practice

2 What is Node.js?

Node.js is "an asynchronous event-driven JavaScript runtime, designed to build scalable network applications"[3].

But what does that mean?

- **Asynchronous (Async/Await):** "The async/await pattern is a syntactic feature of many programming languages that allows an asynchronous, non-blocking function to be structured in a way similar to an ordinary synchronous function" providing "opportunities for the program to execute other code while waiting for a long-running, asynchronous task to complete."[4]
- **Event driven:** "In computer programming, event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads."[5]

The Node.js runtime is built on Chrome's V8 engine. V8 is a widely used JavaScript engine, used in Chrome (and other Chromium based browsers). V8 compiles and executes JavaScript source code, handles memory allocation for objects, and garbage collects objects it no longer needs. JavaScript is most commonly used for client-side scripting in a browser, being used to manipulate Document Object Model (DOM) objects for example. The DOM is not, however, typically provided by the JavaScript engine but instead by a browser. So V8, as a pure JavaScript engine, does not include any methods for DOM manipulation. V8 does, however, provide all the data types, operators, objects and functions specified in the ECMA standard. https://node.green/ shows exactly which ES2015 (ES6) features are supported by each node version, from this we can see that versions 6.4.0 and above have the best support (though it is recommended to use current LTS version from Node.js).

There are other runtimes for JavaScript applications, often being specifically designed to work with existing Node.js projects, such as Bun[6] that have been created to provide specific benefits. Throughout this course we will only be discussing and using Node.js, however for those interested in web development looking at some alternatives can help give a better understanding on what these solutions provide.

2.0.1 Node Package Manager (NPM)

NPM should be thought of as two separate, though interconnected things:

- Primarily, it is an online package repository for Node projects with well over 1 million packages and growing.
- However, it is also a command line tool used by developers for interacting with the aforementioned repository that handles package installation, version management, dependency management and more.

2.1 Working with Node.js

The seasoned JavaScript developers amongst you may already have a preferred workflow for writing web applications. For those of you who do not, then we recommend using JetBrains' WebStorm IDE. WebStorm is simple to use and is already installed on the lab machines. For any of you wanting to use WebStorm on your personal machines you can get a free education license using your university email.

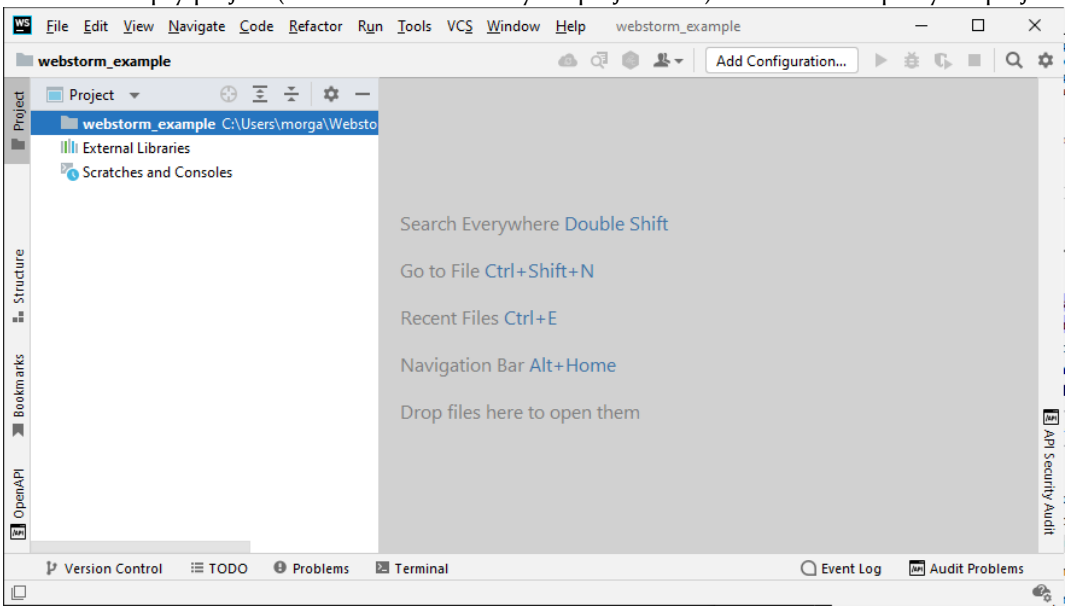
2.2 Running scripts in WebStorm

Running scripts in WebStorm is easy:

1. Open WebStorm.

[3]Taken from: https://nodejs.org/en/about/
[4]Taken from https://en.wikipedia.org/wiki/Async/await
[5]Taken from: https://en.wikipedia.org/wiki/Event-driven_programming
[6]See: https://bun.sh/

2. Select **Create New Project**.
3. Create an empty project (make sure to rename your project first). The IDE will open your project directory.



4. Right click on your project directory and select: **New > JavaScript File**.
5. When prompted for a filename, call it *test.js*.
6. Your new file will open, add the following code from Listing 1.

```
1 console.log('Hello!')
```

Listing 1: A simple console.log JS script

7. Run your script by **right-clicking** and clicking **Run 'test.js'** or using the shortcut SHIFT+F10. **Alternatively:** you can run your scripts by opening your terminal (View > Tool Windows> Terminal) and entering the command `node test.js`.

For more information refer to WebStorm's online documentation[7].

Note: You may need to set up WebStorm to validate ES2015 (ES6) code. To do this go to File > Settings > Languages & Frameworks > JavaScript and set the JavaScript Language Version to **ECMAScript6**.

2.3 Download and Installation (If using a personal computer)

1. Download the recommended version of Node from `https://nodejs.org/en/`
2. Run the downloaded file and navigate through the wizard.
3. Confirm the installation by opening your terminal and typing `node`. You should be presented with the node prompt `>`.
4. Confirm that node has been installed correctly by moving onto Exercise 1.

2.4 Exercise 1: Hello World

1. Create a file named *exercise1.js*.
2. Open the file and insert the code from Listing 2.

```
1 /* Hello, World! program in node.js */
2 console.log('Hello, World!');
```

Listing 2: A simple console.log JS script

3. Open your terminal, navigate to the directory where your file is and run the command `node exercise1.js`.
4. If node was successfully installed, "Hello World!" should be printed to the console.

[7] See `https://www.jetbrains.com/help/webstorm/getting-started-with-webstorm.html`

2.5 Exercise 2: Write a Web Server

In Node, we load modules into other modules using the **require** directive, similar to how Python uses **import**.

1. Create a new file called *exercise2.js*.
2. Use the `require` directive to import the `http` module into a variable for later use.

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

Listing 3: Importing http

Note: You may see WebStorm put a yellow underline on the require function and on hover say "Unresolved function or method require()". This means you need to enable Node.js coding assistance. To do this, go to: File > Settings > Languages & Frameworks > Node.js and NPM and tick the "Coding assistance from Node.js" box.

3. Now use the `http` module's `createServer` function to create a new server. The example code in Listing 4 ignores the content of any request given and instead will just return a 200 OK response with the text "Hello World".

```
http.createServer((request, response) => {
  // Send the HTTP header
  // HTTP Status: 200 (OK)
  // Content Type: text/plain
  response.writeHead(200, {'Content-Type': 'text/plain'});
  // Send the response body as "Hello World"
  response.end('Hello World\n');
}).listen(8081);
console.log('Server running at http://127.0.0.1:8081/');
```

Listing 4: Basic http server with createServer

The documentation for `createServer` can be found here[8].

4. Open your terminal and run `node exercise2.js`.
5. Open your browser and navigate to `http://localhost:8081/`.

2.6 Exercise 3: Handling URL Parameters

Next we want to be able to retrieve URL parameters so that we can use them in our application. Wikipedia provides an explanation of what URL parameters are and their syntax[9].

1. Make a copy of *exercise2.js* and call it *exercise3.js*.
2. Use the `require` directive to import the **URL class** from the **url module**. This module allows us to easily parse URLS.

```
const http = require('http');
const URL = require('url').URL;
```

Listing 5: Http and url imports

3. Next, edit the contents of your `createServer` function. Now whenever we receive a request, we will parse the URL to retrieve its `searchParams`. We will then return a 200 HTTP response with the parameters we found in the response body. Essentially we are "parroting" the request back to the user.

```
http.createServer((request, response) => {
  const url = new URL(request.url, 'http://localhost');
  const parameters = url.searchParams;
  // Write the response
  response.writeHead(200, {
  'Content-Type': 'text/plain'
  });
```

[8] See `https://nodejs.org/api/http.html#http_http_createserver_requestlistener`
[9] See `https://en.wikipedia.org/wiki/Query_string`

```
  response.end(`Here is your data: ${parameters}`);
}).listen(8081);
```

Listing 6: Retrieving url search parameters

4. Again, run your server and then go to your browser and navigate to `http://localhost:8081/`. Add parameters to your URL and the should be shown in your browser (e.g. `http://localhost:8081/?name=Jake&age=35`).

2.7 Exercise 4: Putting it all into Practice

In this exercise, we will create a server that contains a shopping list. When a user navigates to the server with the parameter `itemNum` set, the server will respond with the name of the item at the index of `itemNum`.

1. Create a new file called *exercise4.js*.
2. Import the `http` and `url` modules.
3. Create a constant that contains a list of item names that are typical of a shopping basket (e.g., milk, bread, eggs, flour, ...).
4. Create a server, and inside it parse the URL for parameters. **Note:** To get a specific parameter by its name we can use `parameters.get('param_name')`, you can think of this much like a dictionary from Python.
5. Find out how you can get the value of a specific named parameter (hint: you can select any variable and press **CTRL+Q** to find out its inferred type. You can also **CTRL+Left-Click** to jump to its definition and find out more about it). The Web API docs[10] will also show you how to do this.
6. Use this to find the value of `itemNum`, and retrieve the parameter's value, i.e. the `name` of the item in the list at position `itemNum`.
7. Return a 200 HTTP response that sends back to the browser the value found above.
8. Test your server by going to `http://localhost:8081/?itemNum=2`. It should print out something along the lines of "You selected item 2: eggs".

Now that we have created a simple web application, let's look at creating out first API.

3 Data Formats (JSON)

Throughout this course, we will be using JavaScript Object Notation (**JSON**). JSON is a lightweight data-interchange format that (as the name suggests) has native support in JavaScript. W3Schools provides a good introduction to JSON if you are unfamiliar[11].

Later in this lab we will introduce the JSON file shown in Appendix A. Notably is JSON an object is represented as key: value pairs enclosed in curly braces.

4 What is ExpressJS?

"Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications."[12]

Since Express is a framework much of the code is already written, making it easy to get a web server up and running. Express is the defacto JS backend framework, consistently being ranked highly among other back-end frameworks from other languages, the popularity of the framework also means we can rely on time-tested functionality and large amounts of documentation and tutorials.

[10] See `https://developer.mozilla.org/en-US/docs/Web/API/URL/searchParams`
[11] See `https://www.w3schools.com/js/js_json_intro.asp`
[12] Taken from `https://expressjs.com/`

4.1 Exercise 5: Introduction to ExpressJs - Hello World!

1. First create a directory named *lab1_ex5*, navigate to this directory in your terminal.
2. Install express using the command `npm install express`.
3. In the *lab1_ex5* directory create a file name *app.js* and add the code from Listing 7.

```
const express = require('express');
const app = express();
app.get('/', (req, res) => {
 res.send('Hello World!');
});
app.use((req, res, next) => {
 res.status(404)
 .send('404 Not Found');
});
app.listen(3000, () => {
 console.log('Example app listening on port 3000!');
});
```

Listing 7: Basic express app

4. in your terminal, run `node app.js`.
5. Navigate to `http://localhost:3000/` to see the output.

The app starts a server and listens on port 3000 for connections. When receiving a HTTP request for the root `/` resource, the app responds with "Hello World!". For every other path, it will respond with a 404 Not Found HTTP response.

Note: You can read more about HTTP ports here[13].

5 RESTful APIs

REpresentational State Transfer (REST) is an architecture that makes use of the HTTP protocol for communicating between different clients and servers on the Web. A REST API provides access to resources that the client can access and modify using the HTTP protocol. Each resource is identified using URIs and unique identifiers. Table 1 below shows how different HTTP methods are used in REST to perform specific actions.

HTTP Method	REST ACTION	Example URL	Example (A blog)
GET	Retrieve a resource	/articles/1234	Viewing a blog article
POST	Add a new resource	/articles/	Posting a new blog article
PUT	Replace an existing resource	/articles/1234	Updating (through replacement) a blog article
DELETE	Delete a resource	/articles/1234	Delete a blog article
PATCH	Partial update of a resource	/articles/1234	Updating a part of blog article

Table 1: An example of how different HTTP methods are used within an application.

The difference between PUT and PATCH is sometimes confusing. Both are used for updating data, but PUT will update the whole resource while PATCH updates by identifying only the parts that need to change, similar to a diff operation.

5.1 Exercise 6: Basic Routing in Express

In Express, we can use these HTTP methods to perform different actions. In this exercise, we will use Express to make a single web page react differently depending on the HTTP method used.

1. Create a new directory named *lab1_ex6*. Inside this directory install express as before using the command `npm install express` and create an *app.js* file.

[13]See `https://en.wikipedia.org/wiki/Port_(computer_networking)`

2. As with exercise 5, import the Express module and create an express server using a variable called `app` with the code in Listing 8.

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

Listing 8: Retrieving url search parameters

3. Add a function to handle GET requests to the application root. This GET request should just return the string "HTTP request: GET /" as shown in Listing 9.

```
app.get('/', (req, res) => {
  res.send('HTTP request: GET /');
});
```

Listing 9: Retrieving url search parameters

4. Next, create similar functions for **POST**, **PUT**, and **DELETE** requests that respond with the appropriate message.
5. Call the `listen` function at the end to start a server on port 3000.
6. Run the server.
7. To test your simple API we recommend CURL[14]. Postman is another option but we will be looking into this in the next lab (however those of you with experience, or a dislike of the terminal it may be a good option).
   - CURL works in the terminal so open a terminal and try the command below to fetch all the users
     `curl localhost:3000/users -X GET`
     Note: The `-X` flag defines what HTTP method we want to use. Other important tags are `-d` for supplying data, and `-H` for setting headers (notably we will need to set the Content-Type header to application/json when sending JSON in the body of our request). For now simply play around with using the different HTTP methods.

5.2 Exercise 7: Creating an API - Micro-blogging Site

Now that we understand the basic concepts behind Express, we can create our first API. We will create an API for a micro-blogging site (like Twitter), beginning with the following functionality:

URI	Method	Action
/users	GET	List all users
/users/:id	GET	List a single user
/users	POST	Add a new user
/users/:id	PUT	Edit an existing user
/users/:id	DELETE	Delete a user

Table 2: Basic API specification for user endpoints.

1. Create a new directory called *lab1_ex7*. Inside this directory install Express as we have done before and create an *app.js* file.
2. Once again, within *app.js* import the `express` module and create an Express server.
3. Copy and paste the JSON from Appendix A into a file called *users.json* in the same directory.
4. Import the list of JSON users into your application using the require directive as shown in Listing 10. You can confirm it worked by printing it to the console.

```
const data = require('./users.json');
const users = data.users;
console.log(users);
```

Listing 10: Importing users from JSON file.

[14] See `https://curl.se/docs/manual.html`

5. Now we can start building the API functionality. Create the list all users function that returns the JSON as shown in Listing 11.

```
app.get('/users', (req, res) => {
  res.status(200).send(users);
});
```

Listing 11: Listing all users endpoint.

6. Creating the function to list one specific user is more difficult as we have to retrieve the correct user from the list as shown in Listing 12.

```
app.get('/users/:id', (req, res) => {
    const id = req.params.id;
    let response = `No user with id ${id}`;
    for (const user of users) {
        if (parseInt(id, 10) === user.id) {
            response = user;
            break;
        }
    }
    res.status(200).send(response);
});
```

Listing 12: GET specific user endpoint.

7. Now we will write the POST request. This time the request will contain a JSON object that contains the data required for the new user. Before doing so, we need to include the **body-parser** module (included with Express) - this will take care of interpreting request bodies as JSON for us. Import it and then tell the express `app` object to use it as seen in Listing 13.

```
// Import should be at the top of the file
const bodyParser = require('body-parser');
// Tell the express app to expect json in the body of the request (must be before any routes)
app.use(bodyParser.json());
```

Listing 13: Adding body parser to our Express API.

8. Now we can write our POST function as shown in Listing 14.

```
app.post('/users', (req, res) => {
    const newUser = req.body;
    users.push(newUser);
    res.status(201) // POST requests should return 201 if they create something
        .send(users);
});
```

Listing 14: POST user endpoint.

9. Using the 'list one user' and the 'add new user' functions as a template, we can create a function for the PUT method shown in Listing 15.

```
app.put('/users/:id', (req, res) => {
    const id = req.params.id;
    const updatedUser = req.body;
    for (const user of users) {
        if (parseInt(id, 10) === user.id) {
            const index = users.indexOf(user);
            users[index] = updatedUser;
            break;
        }
    }
    res.status(200)
        .send(updatedUser);
});
```

Listing 15: PUT user endpoint.

10. Finally, to create our DELETE method, we use the JavaScript array `splice` method to remove the user from the list as shown in Listing 16.

```
app.delete('/users/:id', (req, res) => {
    const id = req.params.id;
    for (const user of users) {
        if (parseInt(id, 10) === user.id) {
            const index = users.indexOf(user);
            users.splice(index, 1); // Remove 1 user from the array at this index
        }
    }
    res.send(users);
});
```

Listing 16: PUT user endpoint.

11. Make your application listen on port 3000 and test using Postman or CURL. Check that all 5 endpoints work as expected.
    Note: An example CURL request for posting a user is shown below, you can use the same request for PUT (making sure to change the HTTP method set with the `-X` flag)
    `curl localhost:3000/users -X POST -H "Content-Type: application/json" -d "{\"id\": 1010, \"age\":25, \"first_name\":\"Jane\", \"last_name\": \"Doe\", \"gender\": \"other\", \"email\": \"jd@gmail.com\"}"`

Note: The remaining exercises have been left vague intentionally; how you implement them is up to you, though feel free to discuss your ideas with tutors.

5.3 Exercise 8: Adding Followers

For this exercise, make a copy of your solution to exercise 7, renaming it *lab1_ex8*. Then add functionality that allows users to follow other users. The JSON for each user should include a `following` key, the value of which is a list of user IDs that the user follows.

Make sure you add the following:

1. Add the followers feature to the project, by copying and updating the existing functionality.
2. Add a **follow** function to the API. This function should add a new user ID to a specified users following list.
3. Add an **unfollow** function.
4. Add a **view following** function. This should show all the users that a specified user is following.

5.4 Exercise 9: Adding Micro-blog Posts

This exercise is optional, if you feel comfortable with JavaScript and the content covered feel free to move on Lab 2 at this point. However, do note the difficulty increases substantially as we work with many new technologies.

Each user should also have a list of micro-blog posts. Functionality should exist that:

- Creates a new post for a user
- Retrieves all of a user's posts
- Retrieves a single post of a specified user
- Updates an existing post for a user
- Deletes an existing post
- Retrieves all the posts from all the followers of a specified user
- **(Extra for experts)** Implement 'likes' on posts

6 Final Words

We have now written our first API using Node. However, a problem with our application is that the data isn't persisted anywhere. Once the server is stopped (or crashes), we will lose all of the changes made during the lifetime of the API.

In the next lab we look at persistence and many other best practices (including introducing TypeScript) for creating fully fledged APIs using Node.

A users.json

The JSON data below was generated using an automatic online data generator[15]. We use this data in the exercises in this lab as an example.

If you have problems copying and pasting this text, a standalone file is available on Learn


```json
{
    "users": [
        {
            "id": 1001,
            "age": 35,
            "first_name": "Burch",
            "last_name": "George",
            "gender": "male",
            "email": "burchgeorge@geofarm.com"
        },
        {
            "id": 1002,
            "age": 31,
            "first_name": "Rachelle",
            "last_name": "Chang",
            "gender": "female",
            "email": "rachellechang@geofarm.com"
        },
        {
            "id": 1003,
            "age": 38,
            "first_name": "Sheri",
            "last_name": "Bennett",
            "gender": "female",
            "email": "sheribennett@geofarm.com"
        },
        {
            "id": 1004,
            "age": 32,
            "first_name": "Fisher",
            "last_name": "Dillard",
            "gender": "male",
            "email": "fisherdillard@geofarm.com"
        },
        {
            "id": 1005,
            "age": 20,
            "first_name": "Pope",
            "last_name": "Bailey",
            "gender": "male",
            "email": "popebailey@geofarm.com"
        }
    ]
}
```


Listing 17: Example users JSON file

[15] https://json-generator.com/